Universidad de Alcalá
Escuela Politécnica Superior

Máster Universitario en Ingeniería de Telecomunicación

Trabajo Fin de Máster

Sistema Autónomo de Teleasistencia Robótica mediante técnicas de Inteligencia Artificial

Autor: Diego López Pajares

Director: María Dolores Rodríguez Moreno

Codirector: Pablo Muñoz Martínez

2016

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Máster Universitario en Ingeniería de Telecomunicación

Trabajo Fin de Máster

Sistema Autónomo de Teleasistencia Robótica mediante técnicas de Inteligencia Artificial

Autor: Diego López Pajares

Director: María Dolores Rodríguez Moreno

Codirector: Pablo Muñoz Martínez

Tribunal:

Presidente: Manuel Rosa Zurera

Vocal 1º: José Antonio Portilla Figueras

Vocal 2º: María Dolores Rodríguez Moreno

Calificación: .....

Fecha: .....

"El esfuerzo es solo esfuerzo cuando comienza a doler"
José Ortega y Gassett.

Agradecimientos

La gratitud da sentido a nuestro pasado, trae paz al presente y crea una visión para mañana.

Anónimo.

Me gustaría aprovechar este documento para agradecer a todas las personas que con su apoyo han hecho posible la consecución de este Trabajo Fin de Máster.

En primer lugar me gustaría agradecerle a la Dra. María Dolores Rodríguez Moreno su apuesta por mí para realizar este proyecto, gracias al cual he mejorado y madurado como persona, enfrentándome a nuevos retos que me han aportado experiencias inolvidables. También me gustaría darle las gracias al Dr. David Fernández Barrero ya que también apostó por mí cuando comenzó esta aventura, enseñándome además que la vida sin una interfaz gráfica es posible.

No podía dejar de escapar la oportunidad de darle las gracias a Pablo Muñoz Martínez, por su paciencia a la hora de resolverme las dudas y por los conocimientos técnicos que me ha aportado. También se merecen una mención en este apartado los otros compañeros de laboratiorio: Dani, Fer y Rubén, ya que gracias a ellos y a Pablo, se han hecho más entretenidas las horas de trabajo.

Por último le agradezco a mi familia todo su apoyo tanto moral como económico a lo largo de estos años de carrera y de máster, cuyo fruto ve ahora la luz.

A todos, muchísimas gracias.

Resumen

El objetivo de este proyecto es implementar una arquitectura de control autónoma en un robot comercial orientado a ofrecer servicios de asistencia domiciliaria. Para ello, la arquitectura de control debe ser capaz de generar un plan que cumpla con los objetivos fijados por el usuario, permitiendo la movilidad por todo el hogar. Además, dicha arquitectura de control debe monitorizar el estado de ejecución del plan asegurando la integridad del robot mediante la verificación de la ejecución de las acciones planificadas en un entorno dinámico, consiguiendo un grado de autonomía aceptable para la aplicación que se quiere conseguir.

Palabras clave: Inteligencia Artificial, Deliberador, Ejecutor, Capa funcional, Sistema operativo robótico.

Abstract

The aim of this project is to implement an autonomous control architecture in a commercial robot with the objective of providing home care services. This control architecture should be able to generate a plan that meets the targets set by the user, allowing mobility throughout the home. In addition the control architecture should monitor the execution status of the plan, ensuring the integrity of the robot by verifying the implementation of the actions planned in a dynamic environment, achieving an acceptable degree of autonomy for the application to be achieved.

Keywords: Artificial Intelligence, Deliberative layer, Executor layer, Funcional layer, Robot operating system.

Resumen extendido

Las sociedades más desarrolladas están experimentando un envejecimiento de la población como consecuencia del aumento de la esperanza de vida y la disminución en las tasas de natalidad. Dicha afirmación la contrasta el Instituto Nacional de Estadística: según sus informes, el 18,2 % de la población española está compuesta por personas cuya edad supera los 65 años. Dicho porcentaje aumentará paulatinamente hasta alcanzar el 24,9 % en el año 2029, indicando las previsiones que en el año 2064 el 38,4 % de la población estará comprendida por personas de la tercera edad.

Este envejecimiento de la población tiene como resultado un aumento del número de personas que, debido a los achaques propios de la edad, ya sea por enfermedades o bien por limitaciones físicas, se ven impedidas. Dichas adversidades tienen secuelas nefastas tanto para el anciano como para sus familiares, que tienen que dedicar la mayor parte de su tiempo libre en atender y cuidar a sus seres queridos. Tal es la sobrecarga emocional y física que sufren estos familiares que pueden llegar a desarrollar un trastorno conocido como *Síndrome del cuidador*, el cuál se caracteriza por un agotamiento psíquico y físico que afecta a su calidad de vida. Bajo esta situación se encuadra el concepto de dependencia, definido por el Consejo de Europa como la necesidad de ayuda o asistencia importante para las actividades de la vida cotidiana.

Con el objetivo de atender a este colectivo, y evitar así situaciones de soledad, abandono y episodios de emergencia, nace la teleasistencia. La teleasistencia en su versión más básica consiste en un dispositivo electrónico compuesto por un botón de alarma diseñado para utilizarse a modo de pulsera o de colgante. Ante cualquier situación anómala el dependiente pulsa el botón y dicho instrumento se conecta telefónicamente con un centro de atención especializado que se encarga de resolver la urgencia. En la actualidad estos servicios de teleasistencia los proporcionan empresas como Cruz Roja o Atenzia cobrando una cuota mensual a sus clientes. Sin embargo este dispositivo necesita una respuesta activa por parte del dependiente, ya que es necesario pulsar el botón de alarma para conectar con el operador sanitario, por lo que ante desmayos o pacientes reacios a su utilización resulta inútil.

Este Trabajo Fin de Máster pretende mejorar el sistema de teleasistencia anteriormente mencionado construyendo una arquitectura de control autónomo para un robot comercial que consiga suplir las carencias presentes en los servicios de teleasistencia actuales. Con este robot se pretenden atender situaciones de peligro en personas dependientes gracias al uso de técnicas de Inteligencia Artificial (IA), consiguiendo que el robot sea capaz de dirigirse de forma autónoma a distintas dependencias de un hogar. Además será realizará fotografías y establecerá comunicación con un operador sanitario que se hará cargo de la situación, eliminando así la dependencia activa por parte del paciente.

La arquitectura encargada de realizar este cometido es MoBAr, una arquitectura de control autónomo compuesta por 3 niveles de abstacción cuya estructura se describe a continuación:

- **Deliberador:** capa superior encargada de generar planes de alto nivel que alcancen los objetivos fijados por el operador, utilizando un modelo del entorno que rodea al robot junto con un modelo

de éste.

- **Ejecutor:** capa intermedia que ejecuta los planes de alto nivel traduciendo sus órdenes en acciones de bajo nivel entendibles por la capa funcional. Además realiza una supervisión de la ejecución del plan para mantener al robot dentro de los márgenes nominales de operación.
- **Capa funcional:** capa inferior que interactúa con los sensores y actuadores del robot. Está compuesta por acciones simples que definen la movilidad y funcionalidad del robot.

La arquitectura ejecutará los planes de alto nivel de forma autónoma, permitiendo al robot atender alarmas generadas por el sistema externo sin intervención alguna por parte de un operador, resolviendo por sí misma problemas que puedan aparecer durante la ejecución del plan. También permite tomar el control por parte del operador para comprobar el estado de distintos sensores y/o modificar el plan incluyendo o quitando objetivos a cumplir.

El proceso de implementación de la arquitectura de control siguió una secuencia lógica de acciones cuya descomposición se detalla a continuación:

1. Búsqueda de un robot comercial.
2. Desarrollo de la capa funcional: utilización un framework de desarrollo robótico compatible con el robot para programar la lógica de bajo nivel integrada en la capa funcional.
3. Desarrollo del ejecutor: creación de adaptadores que comuniquen la capa funcional con el ejecutor y desarrollo de secuencias de ejecución que verifiquen el correcto funcionamiento de los adaptadores.
4. Implementación del deliberador: modelado del robot y su entorno en un lenguaje de planificación de alto nivel e integración del deliberador con el ejecutor a través de un adaptador.
5. Creación de planes complejos que integren las 3 capas de la arquitectura y proporcionen la lógica final que se desea alcanzar.
6. Realización de escenarios de prueba en los que se verifique el correcto funcionamiento de la arquitectura de control.
7. Búsqueda de mejoras.

La consecución de estos pasos obtiene un robot que es capaz de moverse por las estancias de un hogar de forma autónoma, pudiendo esquivar obstáculos inesperados y gestionando de manera inteligente la energía de su batería, mejorando así los servicios de teleasistencia actuales gracias a la gestión de alertas no intrusivas que no necesitan colaboración activa por parte de la persona dependiente.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En este capítulo se hace una breve introducción al desarrollo del Trabajo Fin de Máster, mostrando en primer lugar las motivaciones que han contribuído al desarrollo del mismo. En segundo lugar se exponen los objetivos que se pretenden alcanzar tras el desarrollo completo del trabajo, y por útimo, se muestra el esqueleto de la memoria del proyecto.

1.1 Motivación

Los avances tecnológicos experimentados en los útimos años han provocado una enorme expansión en el campo de la robótica y de la Inteligencia Artificial (IA), hasta tal punto que la robótica en la actualidad está presente en numerosas áreas tecnológicas (automatismos industriales, misiones espaciales, educación, asistencia, entretenimiento, etc), cuyo campo de investigación es uno de los más potentes.

Hasta no hace muchos años, sólamente era posible controlar estos robots mediante teleoperación: un operador con conocimientos específicos en el área era el que enviaba de forma manual los comandos al robot, supervisando en todo momento la actividad del mismo. Sin embargo, la tendencia actual pasa por incorporar IA en los robots, eliminando así la necesidad de supervisión constante por parte del operador. Esto ha sido posible gracias al avance de la IA, mediante la cual se han desarrollado arquitecturas software que dotan a los robots de autonomía propia, con grandes capacidades de movimiento y capaces de analizar toda la información externa recogida por los sensores del robot. Con toda esta información, las capas software evalúan meticulosamente la situación para indicarle al robot la acción o acciones que debe realizar. Este tipo de arquitecturas se han estado utilizando sobre todo en robots de exploración, ya sea para misiones espaciales, o bien para inmersiones submarinas, dónde por problemas de visibilidad, retardo de comunicación entre el robot y la estación de control, y condiciones extremas para los humanos era imposible realizar una teleoperación clásica.

Actualmente en el ámbito espacial existen varias arquitecturas de control autónomo como pueden ser IDEA [1], GOAC [2, 3] o MoBAr [4, 5], cuyo objetivo pasa por enviar comandos a un robot para que los ejecute de forma autónoma. El robot recibe dichos objetivos y crea un plan para poder llevarlos a cabo en base a la información que recibe del exterior, retornando al operador los resultados de la misión. A priori puede parecer que este tipo de arquitecturas tienen un enfoque muy específico, pero dista mucho de la realidad, ya que estas arquitecturas poseen un nivel de abstracción que hace posible su integración en varios ámbitos, entre los que se encuentra la teleasistencia.

La teleasistencia consiste en un servicio de ayuda destinado personas mayores o con discapacidad que les permite pedir ayuda en caso urgencia, cuyo producto actual consiste en un colgante con un botón de

emergencia que al pulsarlo conecta al dependiente con una central telefónica en la que un operador se encarga de resolver el problema. Sin embargo este producto tiene serios problemas ya que habitualmente las personas mayores lo atribuyen a un signo de debilidad y son reacios a ponérselo, por no señalar el caso de los desmayos, en el que la persona dependiente no puede pulsar el botón. Por estos motivos se ha pensado en aplicar este tipo de arquitecturas de control a un robot comercial con el objetivo de que, ante una urgencia, el robot se dirija automáticamente a ver el estado de la persona dependiente. En ese momento se conectará con el teleoperador especializado, transmitiéndole la situación mediante una conexión de audio y vídeo.

1.2 Objetivos

Los objetivos propuestos para la realización de este proyecto son:

1. Buscar un robot móvil con velocidad suficiente para moverse en un entorno doméstico. Además tiene que tener la capacidad de realizar fotografías y vídeo, y tener licencia de código libre. En este sentido se prefiere una solución de bajo coste.
2. Implementar la lógica de bajo nivel que le proporcione al robot capacidades básicas de movimiento, captura de imágenes orientadas, detección de obstáculos y posicionamiento automático en una estación de carga.
3. Integrar dicha lógica de bajo nivel en la arquitectura de control autónoma MoBAr desarrollada por el Grupo de Sistemas Inteligentes (GSI) de la Universidad de Alcalá.
4. Desarrollar los planes del ejecutor adaptados a la aplicación específica para la que se va a utilizar el robot. También se tendrán que implementar los adaptadores que comunican al ejecutor con sus capas adyacentes.
5. Modelar el entorno por el que se mueve el robot.
6. Conseguir que el robot sea capaz de moverse en un entorno dinámico, esquivando obtáculos imprevistos.
7. Realizar pruebas que verifiquen el correcto funcionamiento de la arquitectura de control.

1.3 Estructura y contenidos

En este apartado se muestra a modo resumen la estructura lógica que se va a seguir en la memoria.

- **Capítulo 1:** describe la motivación, objetivos, estructura y contenidos de la memoria.
- **Capítulo 2:** analiza del estado del arte con distintas arquitecturas de control para robots autónomos, sistemas operativos robóticos y robótica orientada a la teleasistencia.
- **Capítulo 3:** desglosa todos los componenetes hardware utilizados en el proyecto.
- **Capítulo 4:** muestra una visión general de la arquitectura de control autónoma implementada en el proyecto.
- **Capítulo 5:** introduce conceptos clave del sistema operativo robótico y describe el desarrollo software de la capa funcional.

- **Capítulo 6:** hace una pequeña introducción teórica del framework de desarrollo implementado en el ejecutor y describe el desarrollo realizado en los adaptadores y el modelado del plan del ejecutor.
- **Capítulo 7:** muestra la estructura de deliberador, el modelado del entorno y la librería de interacción con el planificador.
- **Capítulo 8:** recoge los resultados que evalúan el comportamiento de la arquitectura sobre varios escenarios.
- **Capítulo 9:** valora la solución obtenida y añade mejoras futuras que resuelvan los problemas encontrados.
- **Capítulo 10:** presenta el presupuesto de ejecución del proyecto.
- **Apéndices:**
  - *Apéndice A:* amplia la descripción del sistema operativo robótico utilizado en la capa funcional (instalación, herramientas, simuladores, etc.).
  - *Apéndice B:* muestra el proceso de migración de la arquitectura de control a una plataforma hardware de bajo consumo.
  - *Apéndice C:* muestra la herramienta a través de la cual se han realizado todos los experimentos del capítulo 8.
  - *Apéndice D:* contiene documentación adicional que verifica la asistencia a seminarios y concursos como parte de la asignatura Introdución al Trabajo Fin de Máster.

Capítulo 2

Estado del Arte

En este capítulo se muestra un estudio de las diferentes arquitecturas de control autónomas, así como de los sistemas operativos robóticos existentes en el mercado actual y diversos robots orientados al ámbito de la dependencia.

2.1 Arquitecturas de control autónomas

Desde el origen de los tiempos el ser humano ha construído artefactos o máquinas complejas con el fin de facilitar los trabajos más duros (primeras herramientas paleolíticas, la imprenta, el telar mecánico, automóvil, etc.). Asímismo muchos se han marcado como objetivo construir máquinas semejantes al hombre para que trabajasen por él. En 1495 Leonardo Da Vinci diseñó el primer robot humanoide bautizado como Caballero Mecánico [6], un guerrero con armadura capaz de realizar movimientos similares a los de los humanos. Pero no es hasta 1956 cuando General Motors consigue instalar el primer robot comercial en una de sus cadenas de montaje, cuya patente y diseño pertenecían a George Devol [7]. A partir de entonces la robótica se fue expandiendo gracias al fruto de la investigación, hasta tal punto, que en la actualidad tiene aplicaciones en prácticamente cualquier disciplina (educación, ocio, industria, medicina, misiones espaciales, etc.). Esta explosión ha introducido un nuevo concepto de la robótica, el robot autónomo, ya que en muchas de las aplicaciones actuales es imposible realizar una teleoperación clásica como se había venido realizando históricamente, surgiendo así la necesidad de crear robots que actúen de forma autónoma, con mínima o sin supervisión por parte de un operador. Es aquí dónde la IA ha jugado un papel determinante, permitiendo al robot tomar sus propias decisiones, haciendo uso del paradigma de la inteligencia humana. Precisamente, la Real Academia Española (RAE) define inteligencia como la capacidad para entender, comprender y resolver problemas, conceptos estrechamente ligados con la percepción y la capacidad de recepción, almacenamiento y trato de la información, ideas que R. Murphy [8] empleó para definir las tres funciones elementales que ha de tener un robot autónomo:

- Un sistema de **Percepción** mediante el cual el robot es capaz de obtener datos procedentes del entorno que le rodea. Este sistema está compuesto por un conjunto de sensores que son capaces de proporcionar información útil a otros sistemas.
- Un sistema de **Planificación** que le indique al robot las tareas que debe realizar en base a la información proporcionada por el sistema de percepción y al conocimiento adicional proporcionado por el aprendizaje y a el modelo de actuación.

- Un sistema de **Ejecución** que interactúe con los elementos hardware del robot para realizar las tareas encomendadas.

En función de la relación de cada uno de estas tres funciones elementales y del mecanismo de distribución de la información sensorial se pueden llegar a distinguir 4 tipos de arquitecturas de control:

- Arquitecturas reactivas.
- Arquitecturas cognitivas.
- Arquitecturas híbridas.
- Arquitecturas de control.

2.1.1 Arquitecturas reactivas

Este tipo de arquitectura está inspirada en el comportamiento reactivo que poseen los animales. En 1986 Brooks fue el que puso en práctica esta idea con su arquitectura Subsumption [9], en la que únicamente se tiene en cuenta la percepción actual del entorno, y cuyo objetivo, pasa por que el resultado final sea la contribución de varios comportamientos sencillos.

La figura 2.1 presenta una descripción gráfica de este tipo de arquitectura, en la que aparecen dos niveles, uno de ellos se encarga de recoger toda las información posible de los sensores del robot, y el segundo nivel ejecuta los actuadores correspondientes conforme a los datos recogidos por los sensores.

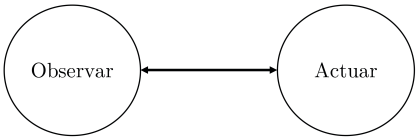


Figura 2.1: Filosofía de la arquitectura reactiva [9]

La ventaja que presenta este tipo de arquitectura es la capacidad de reacción frente a problemas que aparecen en un instante (también conocido como ratio de efectividad). Debido al componente de respuesta rápida, este tipo de arquitectura puede implementarse en sistemas de tiempo real, dónde los tiempos de respuesta ante obstáculos están acotados en un tiempo finito. No obstante esta arquitectura también presenta inconvenientes, ya que reaccionan muy bien ante eventos externos, pero no permite la realización de tareas complejas ya que carece de un sistema de razonamiento.

2.1.2 Arquitecturas cognitivas

El enfoque que presenta la arquitectura cognitiva es diametralmente opuesto al de las arquitecturas reactivas, siendo Nilsson allá por la década de los 60 el primero en hablar de este tipo de arquitectura [10].

La conducta seguida por la arquitectura cognitiva se asemeja más al comportamiento humano, puesto que el robot inicalmente observa el entorno y en base a los datos recogidos planifica las tareas que debe realizar para conseguir los objetivos fijados, siendo leídas y ejecutadas secuencialmente hasta su finalización.

A simple vista se observa que esta nueva arquitectura introduce los conceptos de planificación de larga duración y aprendizaje, imitando el razonamiento humano gracias a la incrusión de un nuevo

componente con respecto al modelo reactivo, la capa de planificación. Esta capa es la encargada de deliberar consecuenteme en función de los datos recogidos por los sensores del robot, generando un plan que debe ser ejecutado por el robot. Sin embargo para que la capa de planificación pueda deliberar correctamente es necesario desarrollar un modelo del entorno lo más real y exacto posible, ya que dicha capa usa este conocimiento para realizar el cálculo de las acciones que se van a llevar a cabo. Además dicho modelo debe ser estable, seguro y consistente para que la generación de acciones sea lo mas precisa posible.

Por lo tanto los sistemas que presentan una arquitectura deliberativa pasan por ser sistemas con un clara estructura jerárquica, en el que el flujo de información fluye desde los niveles superiores a los inferiores y en la que los modelos del entorno que rodean al robot tienen una base de representaciones simbólicas, por lo que consiguen obtener muy buenos resultados en entornos estructurados y predecibles.

Como ejemplo de este tipo de arquitecturas se encuentran ROGUE [11], Soar [12] y NASREM [13].

2.1.3 Arquitecturas híbridas

Por último se encuentran las arquitecturas híbridas, en las que se unen las virtudes de las dos anteriores: la capacidad de deliberar en base a un modelo del entorno, y la capacidad de reaccionar ante estímulos externos. Este tipo de arquitecturas surgieron a principos de los años 90, siendo Ronald C. Arkin el pionero [14], obteniendo un rechazo muy fuerte por parte de la comunidad afín a las arquitecturas reactivas. La filosofía de la arquitectura híbrida se basa en el ciclo sense-plan-act, cuya representación queda recogida en la figura 2.2.

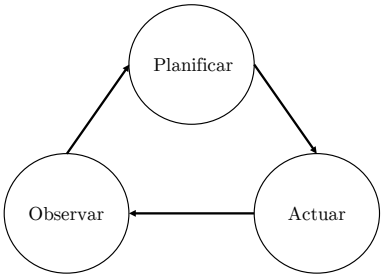


Figura 2.2: Filosofía de la arquitectura híbrida

Al mismo tiempo, las arquitecturas híbridas presentan habitualmente un modelo dividido en capas, cuya estructura típica está compuesta por tres capas o niveles. Mientras que la capa superior se encarga de la planificación, el nivel inferior se encarga de manejar los diferentes sensores y actuadores del robot, siendo la capa intermedia la encargada de supervisar y sincronizar la ejecución de sus dos capas adyacentes. Esta división en capas implica que los tiempos para el procesado de la información en cada una de las capas sea distinto, requiriendo tiempos de respuesta lentos en la capa deliberativa, mientras que en la capa ejecutora los tiempos de procesado deben ser en tiempo real (para poder esquivar obstáculos por ejemplo). Sin embargo en la capa deliberativa es necesario tener conocimiento sobre la duración de la acciones planificadas con el objetivo de comprobar la validez de la información (comprobar que un movimiento no dura más de lo esperado por ejemplo).

Varias son las arquitecturas híbridas que ha sido desarrolladas, entre las que cabe destacar ATLANTIS (desarrollada por el JPL en NASA) [15], 3T (desarrollada por Bonasso) [16], Saphira (desarrollada en el

SRI por Konolige y Myers) [17] o MoBAr [4,5], desarrollada en la Universidad de Alcalá y sobre la cual se construye este proyecto.

La arquitectura MoBAr se basa en una arquitectura híbrida compuesta por 3 capas o niveles de abstracción:

- El deliberador, que es capaz de determinar una trayectoria segura sobre los distintos objetivos que tenga marcados el robot. Está desarrollada sobre la base del *Planning Domain Definition Languaje* (PDDL).
- El ejecutor, que traduce el plan generado por el deliberador y envía las acciones necesarias a la capa funcional. También obtiene datos de la capa funcional para comprobar que el robot está operando correctamente. Esta capa está desarrollada en el lenguaje *Plan Execution Interchange Language* (PLEXIL), desarrollado por el ARC y el JPL de la NASA.
- La capa funcional, que implementa un Sistema Operativo Robótico encargado de comunicarse con los sensores del robot.

En capítulos posteriores se expondrá con mayor nivel de detalle la arquitectura de control MoBAr.

2.1.4 Arquitecturas multiagente

Las arquitecturas multiagente presentan una estructura compuesta por un conjunto de agentes que realizan funciones específicas siguiendo la filosofía *"divide y vencerás"*. Cada uno de los agentes puede involucrar capacidades deliberativas, reactivas o ambas, siendo capaz de resolver problemas complejos gracias a la cooperación entre agentes. Existen dos enfoques en la construcción de sistemas multiagente, un enfoque clásico en el que a cada agente se le dota de la mayor inteligencia posible, y un enfoque constructivo en el que la inteligencia del sistema se consigue con la integración constructiva de todos y cada uno de los agentes.

IDEA [1] es la arquitectura multiagente mas representativa, cada agente tiene encomendados unos objetivos y delibera sobre una parte del modelo, obteniendo la solución final al problema como el conjunto de soluciones obtenidas por cada uno de los agentes. Entre las arquitecturas multiagente también destaca GOAC [2], cuya estructura presenta dos capas: la capa funcional y la capa de deliberación, compuesta por varios agentes que realizan las tareas de deliberación y ejecución.

2.2 Sistemas Operativos Robóticos

El concepto de sistema operativo está ligado a un conjunto de programas que gestionan los recursos hardware, además de proporcionar servicios software. En la última década esta idea se ha ampliado transportándola hasta el ámbito de la robótica con la pretensión de abstraer al programador del hardware del robot.

Los sistemas operativos robóticos son *frameworks* de desarrollo software de robots que guardan numerosas similitudes con el concepto de sistema operativo informático (abstracción de hardware, control a bajo nivel de dispositivos, manejo de paquetes, intercambio de mensajes entre procesos e implementaciones comunes) y que permiten ahorrar tiempo en la implementación de algoritmos debido en gran parte a la reutilización de código. Esta idea surgió debido al enorme grado de complejidad que tiene desarrollar un robot, en el que varios equipos multidisciplinares tienen que trabajar conjuntamente para que la unión hardware-software funcione a la perfección. Por consiguiente, las herramientas que se desarrollan en la

contrucción de un robot, ya sea en el ámbito de la investigación o de la industria, son muy específicas. Exportar estas herramientas al desarrollo de otros robots resulta una tarea árdua y compleja. Por todo esto, y con la idea de reducción de costes, surgió el concepto de sistemas multi-robot, cuya denominación actual son los conocidos Sistemas Operativos Robóticos.

Para tener una visión global de la cantidad de opciones existentes en la actualidad, las siguientes secciones recogen algunos de los sistemas operativos robóticos con mayor transcendencia.

2.2.1 The Orocos Proyect

El proyecto Orocos [18] es un proyecto europero cuyo inicio se remonta al año 2001, y en el que principalmente participan 3 laboratorios de investigación europeos: el laboratiorio belga KULeuven, el laboratorio francés CNRS/LAAS y el laboratorio sueco KTH. La idea de este proyecto pasaba por la implementación de un *framework* de desarrollo software que proporcionase funciones básicas para el control de robots y cuya estructura se descompone en 3 bloques:

- Librería Kinematics and Dynamics (KDL) [19]: simplifica el modelado y la especificación de los movimientos a un problema geométrico que se puede resolver mediante un cálculo matématico sencillo.
- Librería Bayesian Filtering (BFL) [20]: proporciona un conjunto de aplicaciones para el procesado de la información recursiva y estimación de algoritmos basados en la regla de Bayes.
- Librería Orocos Toolchain [21]: contiene un conjunto de herramientas que permiten crear aplicaciones en tiempo real usando módulos software configurables.

En la actualidad *The Source Works* es el principal contribuidor a la infraestructura de tiempo real, mientras que DFKI (Centro de investigación alemán en Inteligencia Artificial) proporciona las herramientas para la generación de código y componentes de *Rock Robotic* [22].

2.2.2 MOOS

Mission Oriented Operating Suite (MOOS) [23] es una plataforma de aplicaciones interconectadas para la operación de robots autónomos muy popular en la comunidad robótica marina. Su creador fue Paul M. Newman en la Universidad de Oxford en el año 2001 con ayuda de investigadores de Oxford y del MIT. MOOS está compuesto por una serie de capas:

- Núcleo de MOOS: arquitectura de comunicaciones basada en redes robustas y de fácil uso.
- Esenciales de MOOS: conjunto de aplicaciones construidas junto al núcleo de MOOS que proporcionan diferentes funcionalidades (logs, manejadores, puentes, etc).

El paquete entero de software que proporciona este framework de desarrollo contiene una gran variedad de aplicaciones y herramientas. Además MOOS tiene una topología en estrella, donde hay un servidor central por el que pasan todos los mesajes de comunicación.

2.2.3 ROS

El origen de *Robot Operating System* (ROS) [24] se remonta al año 2007, en el laboratorio de IA de Standford. Un año más tarde el proyecto se trasladó al Willow Garage, para finalmente en el 2013 acabar en *Open Source Robotics Fundation*. En realidad ROS no es un sistema operativo como tal, sino que es un meta-sistema operativo de código libre, ya que opera sobre sistemas operativos basados en UNIX. Además, está llamado a ser el estándar definitivo para el desarrollo de aplicaciones robóticas debido a su arquitectura distribuida y al elevado número de algoritmos y herramientas que tiene implementadas. De hecho, la industria está adaptando sus productos a ROS, y los investigadores están compartiendo con la comunidad sus ejemplos de código y participando activamente en foros de soporte.

No obstante, ROS proporciona una estructura de comunicaciones distribuida en la que existe un nodo maestro encargado de manejar el intercambio de mensajes entre los otros nodos. De esta forma se consigue una red de comunicaciones distribuida, ya que los nodos de ROS no tienen que estar necesariamente en el mismo equipo (la comunicación de mensajes hace uso del protocolo de comunicaciones TCP/IP). Asímismo, su estructura organizada en paquetes consigue un diseño modular, proporcionando paquetes muy potentes que permiten la reutilización de código (sobre todo para la fusión de datos y para los algoritmos de control).

ROS será el sistema operativo robótico utilizado para desarrollar el proyecto, por lo que en capítulos posteriores se explicará con mayor nivel de detalle.

2.3 Robotica aplicada a la asistencia de discapacitados

Históricamente la robótica ha estado ligada al sector industrial, donde se fue cambiando progresivamente la mano de obra directa por cadenas de montaje robotizadas, que conseguían minimizar los costes y tiempos de producción. Afortunadamente, las capacidades de los robots aumentaron progresivamente apareciendo así nuevas áreas de negocio (hogar, medicina, ejército, entretenimiento, etc.), que resolvían trabajos tediosos y/o peligrosos con un ratio de efectividad más alto que el de una persona. Este aumento de capacidades ha llevado a la industria robótica a explotar nuevos nichos de mercado, como puede ser el ámbito de la ayuda a personas dependientes, desfavorecidas o en riesgo de exclusión social, dónde la integración de un robot puede ser de gran ayuda. Como el presente Trabajo Fin de Máster quiere aprovechar las bondades que ofrece la robótica aplicada a personas dependientes, se van a exponer a continuación algunas de las soluciones robóticas aplicadas a la dependencia existentes en el mercado actual, diferenciando entre robots móviles y robots de ayuda al movimiento.

2.3.1 Robots móviles

Se conoce como robot móvil a los robots que tienen capacidad para moverse dentro de un entorno, siendo perfectos para su uso en un ambiente doméstico, debido a su capacidad de movimiento por toda la casa. Estos motivos han llevado a los robots móviles a ser implementados en aplicaciones orientadas a mejorar la condiciones de vida con personas con capacidad. HOSPI-RIMO [25] (figura 2.3) es uno de estos robots, ha sido diseñado por Panasonic para ayudar a la personas hospitalizadas con el objetivo de que tengan una vida más segura y confortable. El robot puede manejarse por teleoperación o de forma autónoma, y su función pasa por proporcionar un servicio de medicación automática.

Figura 2.3: Robot HOSPI-RIMO

Toyota ha desarrollado el Human Support Robot (HSR) [26] (figura 2.4). Este robot tiene como objetivo ayudar a las personas discapacitadas en las labores cotidianas. Para ello incorpora un brazo robótico que le permite realizar acciones simples pero de vital importancia para personas con movilidad reducida (recoger elementos del suelo, abrir y cerrar puertas, ventanas, etc.). Su control se puede realizar a través de una tablet con una interfaz gráfica o por comandos de voz.

Figura 2.4: Robot Humman Support Robot

La empresa RIKEN con su robot RIBA-II [27] (figura 2.5) es la que proporciona los servicios más avanzados en el ámbito de la asistencia robótica. RIBA-II tiene un objetivo claro, cuidar la espalda de los enfermeros ya que es capaz de levantar a una persona de una silla de ruedas o de una cama. Puede levantar hasta 80 kilos de peso, y puede ajustar el agarre y la brusquedad de los movimientos gracias a las mediciones que realiza su red de sensores. Además es capaz de seguir a una persona gracias a una cámara de reconocimiento facial, o localizar a las personas mediante la captura de tramas acústicas.

Figura 2.5: Robot RIBA-II

También existen alternativas por parte de organismos públicos de la mano de importantes centros de investigación. Uno de los proyectos europeos más importantes y cuyo desarrollo se encuentra ya en fase de pruebas es GiraffPlus [28, 29] (figura 2.6) , un robot que asiste a personas de la tercera edad en sus casas permitiéndoles conectarse con sus familiares, amigos y personal sanitario a través de una pantalla que lleva incluida (telepresencia).

Figura 2.6: Robot GiraffPluss

Este proyecto está desarrollado por un consorcio de universidades europeas entre las que se encuentra la Universidad de Málaga. Además de permitir la comunicación con otras personas, el robot es capaz de detectar situaciones de peligro gracias a la información que le proporcionan una serie de sensores externos distribuidos por la casa (caídas, presión sanguínea, etc.). Sin embargo, el sistema actual tiene un alto coste de producción (4000€ por sistema) que limita su comercialización, aunque en posteriores revisiones tienen previsto disminuir dicho coste.

2.3.2 Robots de ayuda al movimiento

Muchas personas mayores tienen grandes dificultades para moverse ya que sus capacidades físicas se han visto mermadas con el paso de los años. Por este motivo han aparecido robots que facilitan las tareas de movimiento. Panasonic ha creado un cama que es capaz de convertirse en silla de ruedas a través de un simple comando de voz, su nombre: de RoboticBed [30]. Además de poder transformarse, el robot puede ser teleoperado e incorpora unos sensores mediante los cuales es capaz de esquivar obstáculos. Además, la cama incorpora un centro multimedia a través del cual el paciente puede ver la televisión, realizar videoconferencias, o ver las cámaras de seguridad.

Figura 2.7: Robot Roboticbed

Por último, con el fin de ayudar a personas con movilidad reducida en centros comerciales, museos, y otros edificios públicos, se ha desarrollado el andador inteligente C-Walker [31]. Este andador inteligente ha surgido del VII programa Marco de I+D+i de la Unión Europea (Dali) en el que la Universidad de Trento lleva la iniciativa y en la que también participa la empresa española Indra, que se encarga de la integración de prototipo y de la interfaz visual. El objetivo es que el usuario pueda desenvolverse sin peligro en un espacio público guiándolo por el edificio y evitando obstáculos y zonas con aglomeración de gente. Para ello el andador incorpora distintos sensores (presencia, movimiento, presión, etc.) que le permiten mejorar la movilidad e independencia de la persona que lo porta. Se pueden programar los destinos que el usuario desea visitar y el andador de forma autónoma calcula la mejor ruta a la vez que ejerce de guía. Para el guiado, el andador hace uso de distintas tecnologías (códigos QR invisibles, etiquetas RFID y cámaras) que le permiten operar de forma segura por el entorno.

Figura 2.8: Robot C-Walker

Capítulo 3

Plataforma *hardware*

En este capítulo se enumerarán los elementos *hardware* que componen nuestro robot para la aplicación de teleasistencia. La primera sección del capítulo introduce la plataforma robótica con la que se va a trabajar, las siguientes secciones muestran con mayor nivel de detalle cada uno de los elementos que componen el robot. En la última sección se muestra la versión final del robot así como las fases de desarrollo seguidas para su construcción.

3.1 Introducción

Para desarrollar este proyecto se buscó una plataforma robótica móvil orientada a entornos educativos y de investigación que tuviese la agilidad suficiente para desenvolverse con soltura por las dependencias de un hogar. El robot que cumple todos estos requisitos, y que además tiene integración con ROS, es el Turtlebot II. Es un robot móvil de cinemática diferencial que reemplaza a una versión anterior menos sofisticada (TurtleBot), mejorando a su antecesor en la precisión de la medida odométrica, y dotado con mayor velocidad de movimiento y autonomía. La plataforma TurtleBot II está programada con licencias de código libre y posee librerías implementadas que abren el abanico de posibilidades a las que puede ser destinado el robot (desde un robot camarero, construcción de imágenes en 3D, toma de imágenes panorámicas, robot perseguidor, etc.), pudiendo utilizar alguna de esas librerías para este proyecto. A pesar de la imagen que se pueda tener en un principio, TurtleBot II no está compuesto por un solo bloque, sino que consta de varios componentes:

- Base móvil Kobuki.
- Cámara con medida de profundidad Microsoft Kinect.
- Ordenador con sistema operativo Ubuntu.
- Elementos Adicionales.

A continuación se van a desarrollar cada uno de estos bloques para conocer la plataforma *hardware* empleada en el proyecto.

3.2 Base móvil Kobuki

Es el elemento principal que dota de la capacidad de movimiento al robot. En esta base se incluyen motores, sensores, actuadores así como conectores de alimentación bidireccionales (para alimentar los

elementos de la la base o para alimentar elementos externos), y cuyo aspecto (figura 3.1) es similar al conocido robot de limpieza Roomba [32], sólo que orientado a labores de investigación en vez de limpieza, cambiando toda la parte de aspiración por elementos que pemiten ampliar la base. Sin embargo la base Kobuki por sí misma no posee capacidad de procesamiento, y necesita un ordenador externo que mande las órdenes a los actuadores y lea la información de los sensores. En la tabla 3.1 se adjuntan las especificaciones *hardware* de la base.

(a) Vista superior (real)

(b) Vista inferior (esquemática)

Figura 3.1: Base Kobuki

Tabla 3.1: Especificaciones *hardware* de la base móvil Kobuki

Detección de sobrecarga en los motores	Corrientes >3A
Odometría	2 encoders (117 pulsos/mm)
Giróscopo	1 eje (180º/s)
Sensor parachoques	3 (izquierda, centro, derecha)
Sensor de precipicios	3 (izquierda, centro, derecha)
Sensor de baches	2 (uno por motor)
Conectores de alimentación	3 (5V/1A 12V/1.5A, 12V/5A)
Pines de expansión	3.3V/1A, 5V/1A 4 entradas analógicacs 4 entradas digitales, 4 salidas digitales
LEDs programables	2 con tres posibles colores (rojo, amarillo, verde)
Audio	Secuencia de pitidos programables
LED de estado de batería	Verde: nivel de batería ok Naranja: nivel de batería baja Verde parpadeando: cargando batería
Botones	3 botones programables
Batería	2200 mAh (4S1P - pequeña) 4400 mAh (4S2P - grande)
Actualización del firmware	Vía USB
Frecuencia de lectura de los sensores	50 Hz
Adaptador de carga	Entrada: 100-240V AC, 50/60Hz, 1.5A max Salida: 19V DC, 3.16A
Detectores infrarrojos	3 (izquierda, centro, derecha) para la estación de carga
Peso	2,35 kg con la batería 4S1P
Dimensiones	Diámetro: 351,5mm Altura: 124,8mm

La base posee dos motores con unas ruedas dentadas que le proporcionan al robot la capacidad de superar pequeños obstáculos (como por ejemplo alfombras). Estos dos motores tienen dos encoders, que junto con el giróscopo son capaces de calcular la odometría del robot. Además, en la parte frontal del robot hay un parachoques con 3 pulsadores que detectan los obstáculos por colisión. También incluye un detector de precipicios compuesto por 3 sensores de infrarrojos ubicabos en la parte inferior frontal del robot que detectan cambios abruptos en la superficie por la que se mueve el robot. Para detectar posibles baches la base incorpora dos sensores a modo de suspensión en los ejes de las ruedas. Asímismo también posee 3 sensores de infrarrojos en la parte frontal de la base con el fin de posicionar la estación de carga y dirigirse a ella de forma autónoma. También incluye botones y LEDs que pueden ser programados por el ususario. La base permite montar dos opciones de batería (pequeña y grande), siendo la batería de mayor capacidad la elegida para este proyecto, debido a que desde esa batería se alimentarán todos los elementos externos de la base (ordenador, Kinect, pantilt y sensores)

Con esta base, el robot puede moverse con una velocidad máxima de 70 cm/s y una velocidad de rotación de 180º/s, soportando una carga máxima de 5kg y pudiendo detectar acantilados de más de 5 cm. Además proporciona una autonomía de 3 ó 7 horas en función de la capacidad de batería que se le instale (2200mAh o 4400mAh), pudiendo ser capaz de ir a la base de carga automáticamente cuando el robot está situado dentro de una superficie de 2x5 metros frente a la estación de carga. Todas estas especificaciones funcionales quedan recogidas en la tabla 3.2.

Sobre esta base se irán añadiendo elementos adicionales que conformen la estructura final del robot, consiguiendo así todas las funcionalidades que se desean conseguir. El montaje de los diferentes elementos se hará sobre una serie de plataformas adaptadas superpuestas sobre la base.

Tabla 3.2: Especificaciones funcionales de la base móvil Kobuki

Velocidad máxima de traslación	70 cm/s
Velocidad máxima de giro	180º/s
Carga soportada	5 kg
Sensor de acantilados	>5 cm
Tiempo de operación	3/7 h (batería pequeña/grande)
Tiempo de carga	1.5/2.6 (batería pequeña/grande)
Área de carga automática	Superficie de 2x5 m en frente de la estación de carga

3.3 Cámara Microsoft Kinect

Kinect (figura 3.2) fue inicialmente desarrollada por Microsoft para su videoconsola Xbox 360, pero gracias al potencial que tenía, rápidamente la comunidad investigadora empezó a utilizarla para otros fines. Es un controlador que permite interactuar con la consola sin necesidad de contacto físico con la misma gracias a una interfaz que reconoce personas, movimientos, comandos de voz y objetos. Kinect está compuesto por una cámara RGB, un sensor de profundidad y 4 micrófonos que le permiten capturar comandos de voz. Permite hacer cálculos de distancias mediante dos sensores de infrarrojos (emisor y receptor) y obtiene imágenes claras gracias a su cámara RGB. En robótica este elemento tiene una gran utilidad, ya que permite al robot calcular distancias a objetos cercanos con los sensores de infrarrojos y puede reconocer dichos objetos con la cámara RGB, así como aplicar técnicas de *Simultaneous Localization And Mapping* (SLAM) junto con información externa a la Kinect (encoders, giróscopo, etc.). A pesar de tener numerosas funcionalidades, para este proyecto únicamente se va a utilizar la Kinect como cámara de fotos para capturar imágenes en determinadas posiciones de un mapa.

Figura 3.2: Microsoft Kinect

3.4 Ordenador

Para poder analizar toda la información que generan los sensores y poder accionar sobre los actuadores es necesario un ordenador personal que cuente con un sistema operativo Ubuntu. La opción elegida para este proyecto es un ordenador mini-Pc ubicado encima de la base Kobuki, y cuyas características se muestran en la tabla 3.3.

Tabla 3.3: Especificaciones *hardware* del PC empleado

Procesador	Intel Atom (Dual Core)
Memoria RAM	2GB
Sistema Operativo	Ubuntu 14.04
Aplicaciones necesarias	ROS Indigo

Este ordenador será el que tenga instalada toda la arquitectura de control para poder operar con el robot de forma autónoma. Para alimentarlo se ha hecho uso de uno de los conectores de alimentación que proporciona la base Kobuki, en concreto el de 12V/5A.

3.5 Elementos adicionales

Para cumplir con los requisitos del proyecto se han tenido que añadir elementos adicionales a la plataforma Turtlebot II que venía por defecto de fábrica. La figura 3.3 muestra los 3 elementos añadidos y que se explicarán a continuación.

(a) Estación de carga (b) *Pantilt* (c) Ultrasonidos

Figura 3.3: Elementos adicionales del robot

3.5.1 Estación de carga

El primer elemento que se introdujo fue la estación de carga (figura 3.3a), y aunque lo proporcione el fabricante, su adquisición por parte del consumidor final es opcional. Con este módulo el robot puede iniciar la carga de su batería de forma autónoma, ya que la estación de carga posee un LED infrarrojo que emite un haz de luz. Este haz de luz es recibido por los 3 receptores infrarrojos que tiene la base Kobuki en su frontal, pudiendo dirigir el robot hacia la estación de carga en función del punto de mayor luminosidad. Para poder realizar la carga la estación cuenta con dos conectores metálicos que transfieren energía a la batería del robot a través de otros dos bornes metálicos integrados en la parte inferior de la base Kobuki. De esta forma cuando el robot llega a la base, los bornes metálicos entran en contacto y la batería comienza a cargarse.

3.5.2 *Pantilt*

La posición de la cámara es un elemento clave, ya que permite tener información visual sobre la ubicación del robot así como del entorno que lo rodea, y más si el robot está orientado a realizar labores de teleasistencia doméstica. Con la movilidad de la cámara un operador sanitario puede determinar rápidamente la estancia en la que se encuentra el robot y la situación de la persona dependiente sin necesidad de girar el robot para tener una visión panorámica del entorno.

La distribución inicial del robot no contempla una cámara móvil, por lo que se optó por añadir un elemento externo al robot que dotara de movilidad a la cámara. Este nuevo elemento tiene el nombre de *pantilt*, (figura 3.3b) y consiste en unos soportes para la cámara que permiten moverla en los ejes de azimut y elevación. Su construcción ha sido casera, utilizando piezas de aeromodelismo para los soportes y dos servomotores MG996R para realizar los movimientos de precisión.

Servomotores MG996R

Los servomotores son actuadores electrónicos comunmente usados en aplicaciones robóticas. Están compuestos por un motor de corriente continua que gira a gran velocidad, una caja reductora (conjunto de engranajes que reducen la velocidad de giro del motor aumentando su par) y una electrónica de control que permite posicionar el motor en un ángulo de giro específico. El servomotor escogido (MG996R) es la versión mejorada del MG995R, ofreciendo un mayor par y mejor precisión en los moviemientos en parte gracias al rediseño del circuito de control. Su voltaje de trabajo está entre los 4.8 y 6.0 V, pudiendo ofrecer hasta 6 Kg/cm de par. Todas las especificaciones *hardware* de este servomotor se pueden encontrar en la tabla 3.4.

Tabla 3.4: Especificaciones MG996R

Dimensiones	Alto: 40.7 mm Ancho:19.7 mm Profundo: 42.9 mm
Peso	55 g
Torque	9.4 kg/cm (4.8V) 11 kg/cm (6.0V)
Velocidad	0.17s/60º (4,8V) 0.14s/60º (6.0V)
Voltaje de trabajo	4.8-6.6V
Piñonería	Metálica

El control de estos servomotores se realiza a través de una señal *Pulse Wide Modulation* (PWM), una señal rectangular de periodo fijo en la que se modifica el ciclo de trabajo para elegir el ángulo de giro del servomotor. Para entender mejor en funcionamiento de este sistema de control se incluye la figura 3.4, en la que aparece representada la señal PWM asociada al ángulo de giro que tiene el servomotor.

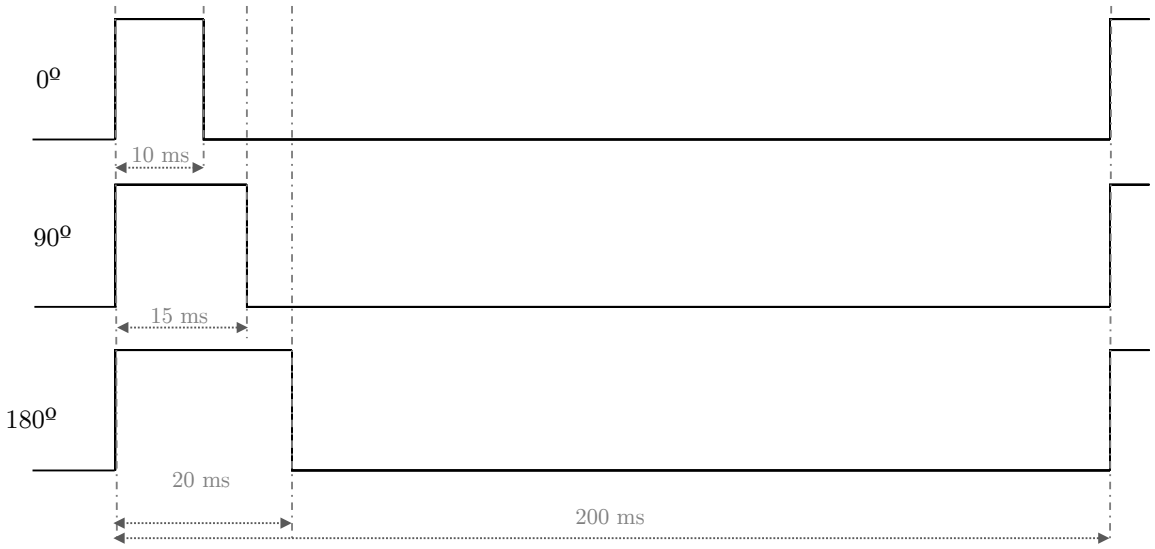


Figura 3.4: Señal PWM empleada para el servo MG996R

La alimentación de los servomotores se realiza a través de uno de los conectores de alimentación que proporciona la base Kobuki, en concreto el conector de 5V, mientras que para generar la señal de control es necesario un microcrontrolador que genere la señal Pulse Wide Modulation (PWM). El microcontrolador escogido para realizar este labor de control ha sido un Arduino Uno.

Arduino Uno

Arduino es una empresa tecnológica de *hardware* libre que diseña, fabrica y vende placas de desarrollo compuestas por un microcontrolador, pines de entrada-salida (analógicos y digitales) y una *Printed Circuit Board* (PCB) que conecta todos los componentes. Toda la plataforma Arduino (tanto *hardware* como *software*) tiene licencias de código libre, por lo que es fácil encontrar librerías programadas en Arduino para sensores de uso común. Además la plataforma Arduino posee un lenguaje de propio programación de alto nivel, cuya sintáxis es similar a C++, que soporta todas las funciones estándar de C y algunas de C++. Para programar sus microcontroladores, Arduino proporciona su propio entorno de desarrollo (Arduino IDE) a través del cuál se programan todas sus placas y que además proporciona una consola serie por la que el programador puede enviar y recibir información del controlador (imprimir datos de sensores, enviar órdenes al microcontrolador, depuración, etc.). De entre todas las placas oficiales que tiene Arduino, la empleada en este proyecto es la placa Arduino Uno, la plataforma más extendida y la primera que salió al mercado, cuyas especificaciones se encuentran recogidas en la tabla 3.5.

Al ser la primera placa que desarrolló Arduino, las especificaciones de la misma están implementadas en casi todas las placas de la compañía. El microcontrolador sobre el que está construida la placa Arduino Uno es un ATmega320 de 8 bits a 16 MHz, con 32 kB de memoria flash de los cuales 0.5 kB están reservados para el bootloader, y que además cuenta con 2 kB de SRAM y 1 kB de EEPROM. En cuanto a las conexiones externas, Arduino Uno tiene 20 pines de conexión, de los cuáles 14 de están destinados a señales digitales, y sólamente 6 estarán dedicados a señales analógicas. Para dotarle de mayor versatilidad a la placa, 6 de los pines digitales tienen la posibilidad de generar señales PWM.

Tabla 3.5: Especificaciones de Arduino Uno

Microcontrolador	ATmega328P
Voltaje de trabajo	5 V
Voltaje entrada (recomendado)	7 - 12 V
Voltaje entrada (límite)	6 - 20 V
Pines entrada/salida digitales	14 (6 de PWM)
Entradas analógicas	6
Corriente pines I/O	30 mA
Corriente pin 3.3 V	50 mA
Memoria Flash	32 kB
SRAM	2kB
EEPROM	1 kB
Velocidad de reloj	16 MHz

Todos estos pines de entrada-salida tienen un voltaje recomendado de trabajo de 5V, aunque las señales de entradas pueden trabajar entre los 5 y 7 V, mientras que las de salida tienen la posibilidad de trabajar a mayores tensiones (de 6 a 20V). En cuanto a la alimentación de la placa, ésta puede ser alimentada a 5 V mediante un pin de entrada, un conector de alimentación o por el puerto USB.

Para realizar el control de la *pantilt*, se van a utilizar dos pines (uno para cada servomotor), en concreto los pines digitales 10 y 11 con salida de señal PWM. La alimentación de la placa se hace mediante la conexión USB, que además sirve como enlace de comunicación entre el microcontrolador y la arquitectura de control montada en el ordenador. De esta forma cuando la arquitectura de control necesite mover la cámara enviará la orden al Arduino a través de la conexión USB, y la placa Arduino se encargará de generar las señales PWM que mueven los servomotores de la *pantilt*.

3.5.3 Sensores ultrasónicos

Los sensores ultrasónicos son sensores de proximidad que detectan objetos en las cercanías del sensor haciendo uso de las propiedades de propagación del sonido. En concreto los sensores de ultrasonidos están compuestos por un altavoz y un micrófono que imitan el sistema de guiado de los muerciélagos. El altavoz emite un pulso de alta frecuencia durante un periodo muy corto de tiempo y activa el micrófono. Dicho pulso se propaga a una velocidad de 343 m/s por el aire (en condiciones normales de temperatura y humedad), rebotando en los objetos que interfieren en su trayectoria, siendo la ondas rebotadas las recogidas por el micrófono. De esta forma se puede conocer la distancia a los objetos haciendo uso de la ecuación 3.1.

$$Distancia\ al\ obstáculo = \frac{Tiempo\ de\ ida\ y\ vuelta \cdot Velocidad\ de\ propagación}{2} \tag{3.1}$$

Este tipo de sensor se puede clasificar como un modelo de radar pulsado, en el que una antena emisora emite una señal que se degrada al propagarse por la atmósfera, y que además se ve modificada al colisionar con los distintos elementos del entorno dependiendo de la sección radar (σ) de los objetos. Una mala configuración de la antera receptora puede provocar que ante esta deformación que sufre la señal original se detecten falsos positivos provocados por el ruido no deseado del sistema, cuya solución pasa por orientar la antena receptora en la misma dirección que la antena emisora y restringir el haz del diagrama de radiación para que la antena sea lo más selectiva posible.

Dentro de la gran variedad en sensores de ultrasonidos se ha escogido el HC-SR04 (figura 3.3c), un referente en sensores de ultrasonidos ampliamente extendido gracias a su relación calidad-precio, con una precisión aceptable a un precio muy bajo como se puede apreciar en las especificaciones técnicas

mostradas en la tabla 3.6. Con este sensor es posible detectar objetos entre 2 cm y 4 metros de distancia, con un ángulo de apertura de la onda sonora de 15º. Además opera a 40 kHz, haciéndolo imperceptible para el ser humano puesto que dicha frecuencia se sitúa en un rango de frecuencias mayor que el rango de audición humano (2 - 20 kHz).

Tabla 3.6: Especificaciones *hardware* del sensor HC-SR04

Tensión de alimentación	5 V
Consumo	15 mA
Frecuencia de trabajo	40 kHz
Distancia mínima	2 cm
Distancia máxima	4 m
Ángulo de apertura	15º
Pulso de disparo	10 μs
Pulso de eco	Señal a nivel alto TTL proporcinal a la distancia medida
Tamaño	40 x 20 x 15 mm
Peso	10 g

El sensor posee 4 pines de conexión, dos de alimentación (*Vcc* y *Gnd*), un pin de *Trigger* que controla el emisor y un pin de *Echo* que recoge los ecos del micrófono para procesarlos como flancos digitales. Para realizar la medición hay que mantener el pin de *Trigger* durante al menos 10 μs a nivel alto. En el momento que el pin de *Trigger* se pone a nivel bajo, internamente el sensor emite 8 pulsos a 40 kHz a través de su altavoz que se propagan por el espacio libre. Cuando el micrófono detecta el eco del pulso transmitido pone a nivel alto el pin de *Echo* durante un tiempo proporcional a la anchura del pulso transmitida. Midiendo con un temporizador el tiempo que la señal de *Echo* se mantiene a nivel alto y con ayuda de la ecuación 3.1 es posible obtener la distancia a los objetos. Para entender con mayor claridad este método, se proporciona un diagrama temporal de las señales de *Trigger* y *Echo* en la figura 3.5.

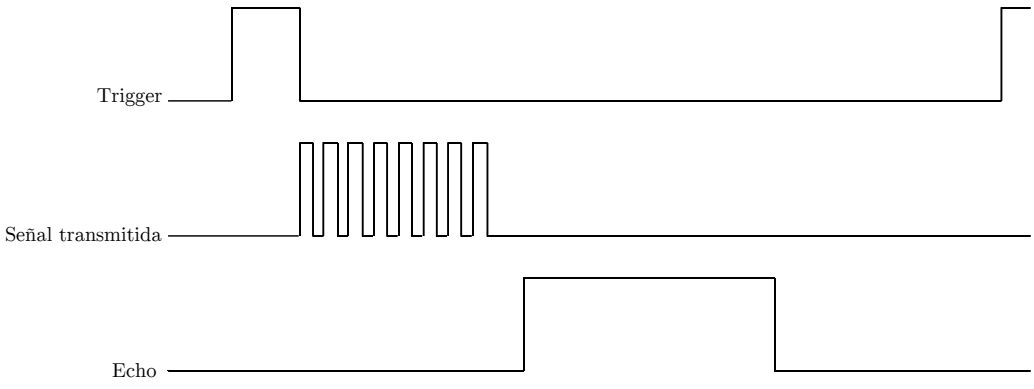


Figura 3.5: Esquema con la señalización del sensor HC-SR04

En cuanto al número de sensores y su ubicación, se pretende cubrir con los mismos una zona de 20 centímetros de ancho a una distancia de 40 centímetros del sensor. Para definir el número de sensores y su colocación primero se ha calculado la zona de cobertura que ofrece un sensor a dicha distancia, y en función del valor obtenido se calcula el número de sensores necesarios. Con ayuda de la figura 3.6 y las

ecuaciones 3.2, 3.3 y 3.4, y sabiendo que $\alpha = 15^{o}$ y $h = 40\ cm$ se obtiene la anchura del haz para un sensor.

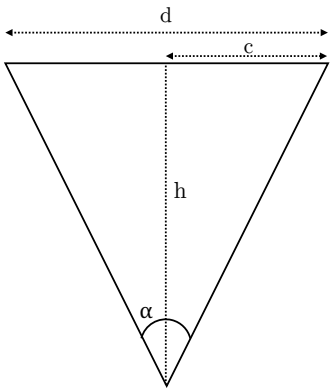


Figura 3.6: Cálculo de la anchura abarcada por el sensor de ultrasonidos

$$\tan\frac{\alpha}{2} = \frac{c}{h} \tag{3.2}$$

$$c = \tan\frac{\alpha}{2} \cdot h \tag{3.3}$$

$$d = 2 \cdot c \tag{3.4}$$

Tras hacer los cálculos se obtiene que la anchura del haz para un sensor HC-SR04 es de 10.53 cm, por lo que para cubrir 20 cm serán necesarios dos sensores de ultrasonidos separados entre sí 10.53 cm, como indica la figura 3.7.

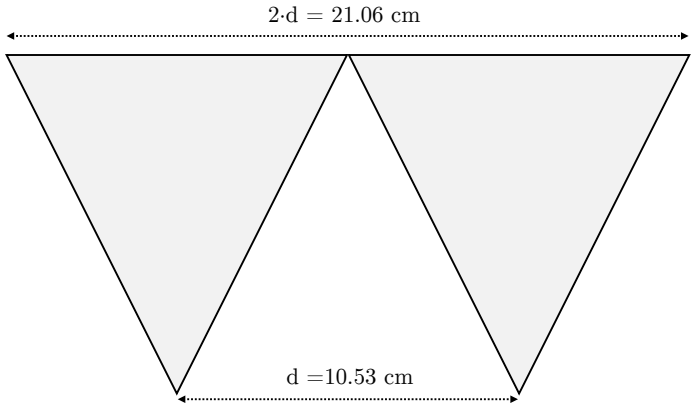


Figura 3.7: Posición de los dos sensores de ultrasonidos

La razón por la que han tomado 40 cm desde el sensor viene determinada por el mapa en el que se moverá el robot. Dicho mapa posee celdas de 40 x 40 cm (como se verá en la sección 8.1), siendo interesante poder detectar obstáculos que se encuentren en celdas adyacentes a las que se encuentre el robot impidiendo su avance. De esta forma se tiene un campo de visión suficiente para detectar objetos comunes en una casa (papeleras, patas de sillas, mesas, muebles, etc.) sin limitar las zonas por las que puede acceder el robot (una zona de visión más amplia impediría que el robot pudiese avanzar por un pasillo estrecho).

3.6 Versión final del TurtleBot II

Para obtener el TurtleBot II es necesario unir los elementos expuestos en las secciones anteriores que incluye este capítulo, necesitándose varias fases hasta contruir la versión final del TurteBot II. No obstante el robot utilizado para este proyecto no se corresponde con el montaje original deI robot debido a los componentes adicionales que se han añadido.

Siguiendo el orden cronológico, las fases para la construcción del robot son las siguientes:

1. Montaje del robot original siguiendo las intrucciones del fabricante para tener una primera toma de contacto. El montaje se puede ver en la figura 3.8a, en la que se observa que el robot se compone de 3 alturas de las cuales solo el estante intermedio queda ocupado.

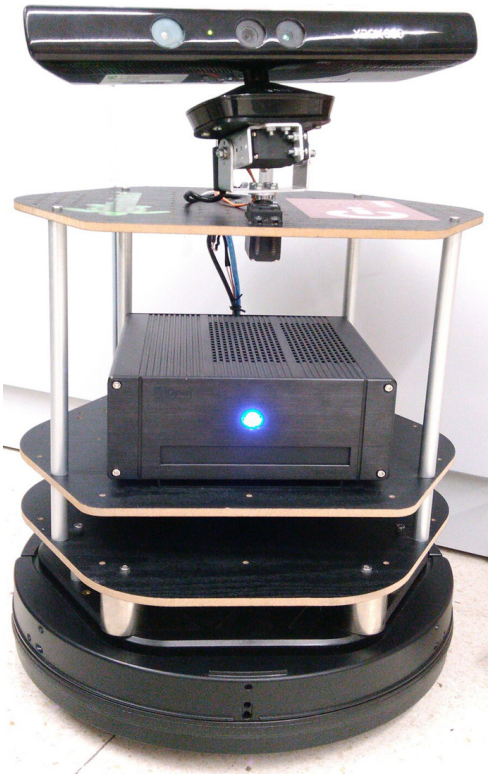
2. Instalación de la estación de carga para probar nuevas funcionalidades.

3. Instalación de un mini-pc en el estante intermedio que facilitase los movimientos del robot. Hasta ahora se controlaba el robot con un odenador portátil conectado a la base Kobuki a través de un cable USB. Para incorporar esta nueva característica fue necesario quitar la Kinect de su posición original.

4. Instalación de la *pantilt*. Se aprovechó el cambio de posición de la Kinect para incluir al robot la *pantilt*, fijando su ubicación en la parte frontal del nivel superior. Se perforó la base superior para incrustar uno de los servomotores de la pantilt con el objetivo de mejorar su estabilidad. Asímismo se camufló el Arduino que controla los servos en la parte inferior de la base superior. Tras añadir y conectar todos los elementos el robot muestra un aspecto distinto con respecto al original, como se puede apreciar en la figura 3.8b.

5. La última modificación que sufrió el robot (figura 3.8c) fue la incorporación de los sensores ultrasónicos. Éstos se incorporaron en el nivel inferior para poder detectar los obstáculos que se encuentran en las proximidades del suelo sin que la señal de los ultrasonidos se refleje en el suelo provocando el indeseado efecto *clutter*.

(a) Montaje original Turtlebot II



(b) TurtleBot II modificado

(c) TurtleBot II modificado (versión final)

Figura 3.8: Turtlebot II en sus diferentes configuraciones

Capítulo 4

Visión general de la arquitectura

En este capítulo se va a describir en qué consiste la arquitectura de control autónoma MoBAr, mostrando una visión global de la misma, así como los bloques básicos que la componen. En capítulos posteriores se abordará con mayor nivel de detalle cada uno de estos bloques.

4.1 Introducción a la arquitectura MOBAR

Model-Based Arquitecture (MoBAr) es una arquitectura de control autónoma de 3 capas (3-T) basada en el paradigma de las arquitecturas de control híbridas. Mediante la descomposicion en tres capas se consiguen crear distintos niveles de abstracción que permiten adaptar el sistema a distintos modelos robóticos, como por ejemplo desde un rover de exploración espacial hasta un robot doméstico autónomo, pasando por un brazo de control robótico o un satélite autónomo.

De las 3 capas, el deliberador, es el encargado de generar un plan de alto nivel que cumpla las metas establecidas por un operador en base a un modelo del entorno que rodea el robot y un modelo del robot. El plan generado establece una trayectoria segura entre los distintos puntos por los que tiene que pasar el robot, siendo capaz de esquivar obtáculos que aparecen en el mapa. Este plan es descompuesto e interpretado por el ejecutor, que se encarga de traducir las órdenes de alto nivel en órdenes simples para enviárselas a la capa funcional. Por su parte, la capa funcional se encarga de ejecutar las acciones simples que recibe del ejecutor, y comprueba si se han ejecutado satisfactoriamente, devolviendo el resultado de la acción al ejecutor. Esta capa funcional es la que está en contacto con el *hardware* del robot, actuando sobre los distintos motores y controladores del robot a la vez que recoge información de sus sensores. En el caso de que se produzca un error al ejecutar algunas de las acciones en la capa funcional, el ejecutor toma el mando para corregir la situación actualizando la información y devolviendo el control al deliberador para que genere un nuevo plan con los datos actualizados.

4.2 Desarrollo de la arquitectura de teleasitencia

En esta sección se va a explicar de forma general el comportamiento de la arquitectura MoBAr orientada a la aplicación de teleasistencia, explicando el comportamiento simplificado de sus 3 niveles.

La capa superior o deliberador contiene el comportamiento cognitivo del sistema, posee información relevante sobre el modelo del robot y el entorno en el que está inmerso, así como la posición inicial del robot y las metas que debe alcanzar. A pesar de contener información del robot, esta capa es independiente

del robot utilizado ya que únicamente modela su comportamiento, sin entrar en detalles de *hardware*, de esta forma es posible cambiar el robot por uno de características similares sin necesidad de realizar ningún cambio en esta capa. Con toda la información almacenada en el deliberador se genera un plan de alto nivel que cumple con los objetivos fijados. El plan posee la ruta que debe seguir el robot para alcanzar sus metas, así como la orientación de la pantilt o acciones que gestionan de forma inteligente la energía de su batería. De esta forma es posible generar un plan que recoja información del hogar de forma autónoma, pudiendo reaccionar con rapidez en situaciones de peligro. Una vez generado el plan el deliberador lo transfiere al ejecutor para su interpretación y ejecución.

El ejecutor se ubica en la capa intermedia de la arquitectura, entre el deliberador y la capa funcional. Es el encargado de coordinar y gestionar el correcto funcionamiento del sistema ya que monitoriza el estado del robot y extrae información del mundo real. Para conseguirlo tiene una comunicación directa con las capas superior e inferior a través de interfaces que traducen las instrucciones de las capas adyacentes.

Por último la capa funcional es la encargada de comunicarse con el *hardware* del robot, actuando directamente sobre él. El ejecutor se comunica de forma directa con esta capa indicándole las acciones que debe realizar (avanzar, girar, hacer foto, ...). Tras ejecutar la acción, la capa funcional devuelve el resultado al ejecutor que determina si el resultado ha sido correcto o no. Si no ha sido correcto, el ejecutor intenta corregir el problema, y si le resulta imposible actualiza los datos para que el deliberador genere un nuevo plan.

Con todo esto, la arquitectura MoBAr proporciona una gran versatilidad, ya que permite cambiar por completo el modelo del robot con tan sólo modificar la capa funcional. Además al ser una arquitectura híbrida es capaz de generar un plan óptimo inicial que puede ser modificado en tiempo de ejecución en el caso de que se produzca algún imprevisto.

4.3 Flujo de ejecución autónomo

Esta sección explica con mayor nivel de detalle el flujo de ejecución de la arquitectura de control MoBAr, teniendo como base el diagrama de flujo representado en la figura 4.1.

El deliberador posee información del mundo que rodea al robot a través de un mapa en el que se representan las zonas posibles de paso y los obstáculos fijos que no puede atravesar el robot, así como puntos posibles de recarga de batería. También dispone de información relativa al robot, indicando la posición en la que se encuentra en el mapa, su nivel de batería, las acciones que puede realizar, energía consumida por acción, etc. Con toda esta información el deliberador actualiza los datos del problema y genera un plan de alto nivel que cumple los objetivos iniciales, esquivando los obstáculos fijos y realizando recargas intermedias en el caso de ser necesario. Este plan de alto nivel es recogido e interpretado por el ejecutor, el cual lee y ejecuta secuencialmente las acciones de alto nivel, haciendo una traducción a órdenes entendibles por la capa funcional. También realiza una supervisión del plan temporizando la duración de los movimientos del robot y comprobando que dichos movimientos finalizan correctamente. Si la acción falla, intenta resolver la situación generando acciones correctoras que reviertan la situación, aunque si no es posible, devuelve el control al deliberador para que genere un nuevo plan. Por último la capa funcional se encarga de ejecutar en el *hardware* las acciones de bajo nivel, efectuando movientos de avance, giro, orientación de la cámara, realización de fotografías, etc. Cuando finaliza la acción devuelve el resultado al ejecutor para que compruebe si se ha realizado correctamente.

Para realizar la comunicación entre capas se han desarrollado dos adaptadores (deliberador-ejecutor y ejecutor-capa funcional) que permiten el entendimiento entre capas. Estos adaptadores se encargan

de realizar lectura de archivos o conversión de datos para que la comunicación pueda fluir por toda la arquitectura.

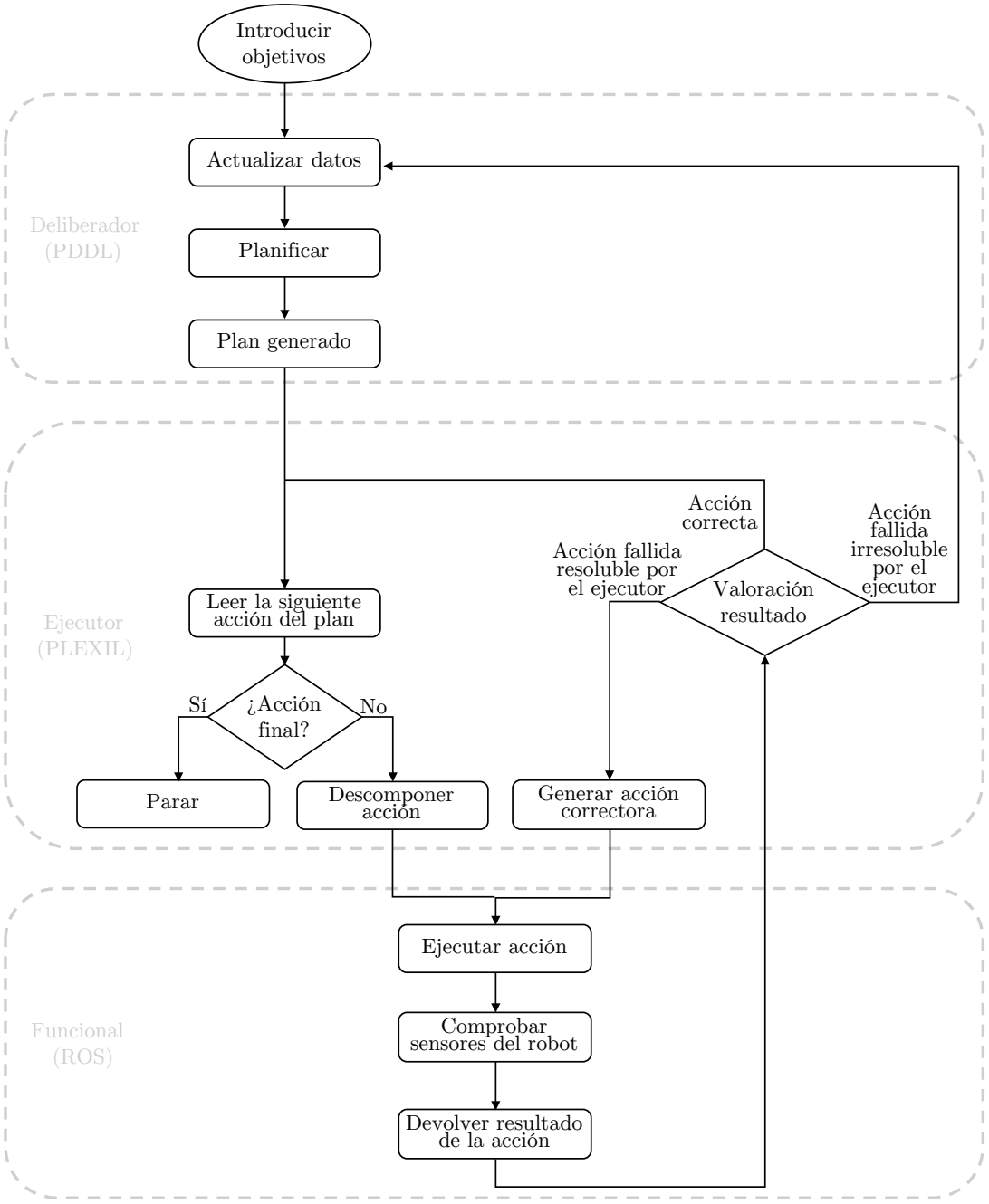


Figura 4.1: Diagrama de flujo de la arquitectura MoBAr

Capítulo 5

Capa funcional

En este capítulo se describe el desarrollo *software* de la capa funcional. Para desarrollar esta capa se ha hecho uso del Sistema Operativo Robótico, ROS. El capítulo comienza haciendo una introducción a dicho sistema operativo, para después estudiar su arquitectura así como sus conceptos claves. Por último se muestra el desarrollo *software* de la capa funcional MoBAr, la cuál hace uso de todos los conceptos explicados a lo largo del capítulo.

5.1 Introducción a ROS

ROS es un *framework* de desarrollo para robots que proporciona la funcionalidad de sistema operativo en clúster heterogéneo, creado en 2007 por el Instituto de Inteligencia Artificial de Standford (SAIL); su visión general se puede concebir como un conjunto de herramientas, librerías y reglas destinados a simplificar el desarrollo robótico en numerosas plataformas *hardware* y *software*. En 2008 el proyecto fue trasladado principalmente al *Willow Garage*, un instituto de investigación robótica con más de 20 instituciones colaborando dentro de un modelo de desarrollo federado. A raíz de de este crecimiento, muchos centros de investigación comenzaron a desarrollar sus prototipos en ROS compartiendo su código. Además, muchas compañías han comenzado a adaptar sus productos para ser compatibles con ROS acompañándolos de modelos de simulación, ejemplos de código y aplicaciones que le facilitan la tarea al programador al adentrarse en una nueva plataforma robótica. En la página web de ROS se pueden encotrar todos los robots comerciales que soportan este sistema operativo robótico [33].

Para ver el modo de operación de ROS, en la siguiente sección se exponen los conceptos clave y la arquitectura típica de este sistema operativo robótico.

5.2 Arquitectura y conceptos claves de ROS

En esta sección se va a analizar la organización de ROS junto con sus conceptos clave para poder entender el desarrollo *software* de la capa funcional. ROS se puede descomponer en 3 niveles básicos:

- Comunicación entre procesos.
- Organización del sistema de ficheros.
- Comunidad de soporte.

El primer nivel pretende explicar la comunicación entre procesos que hace este sistema operativo robótico, así como el manejo de dichos procesos y la comunicación entre varios ordenadores. Por su parte, el segundo nivel explica la estructura de ROS a nivel de carpetas y ficheros, con el objetivo de dar a conocer la estructura de archivos de las aplicaciones generadas bajo este *framework* de desarrollo robótico. Por último, el nivel de comunidad mostrará las herramientas y conceptos disponibles para adquirir y compartir conocimientos en la comunidad de ROS. A continuación se explicarán estos elementos para que el lector se familiarice con el *framework* de desarrollo robótico ROS.

5.2.1 Comunicación entre procesos

ROS crea una red donde todos los procesos están conectados entre sí. Cualquier nodo en el sistema puede acceder a esta red o interactuar con otros nodos, así como ver la información que se está transmitiendo o transmitir información a la red. Para conseguir esto, hace uso de conceptos básicos como maestro, nodo, servidor de parámetros, mensajes, servicios y topics, cuyo análisis se muestra a continuación.

Nodo

Los nodos son procesos que realizan cálculos o cómputos. Estos nodos pueden comunicarse entre sí por la red que crea ROS a través de topics, servicios y servidor de parámetros. Normalmente cada nodo tiene encomendada una función específica, por ejemplo, uno calcula la odometría, otro controla los sensores de choque, otro verifica el estado del robot, separando así funcionalidades entre nodos, lo que hace que el sistema sea modular y mucho más fácil de mantener. Además, dichos nodos se encuentran distribuidos por la red proporcionando una robustez ante puntos únicos de fallo. Esto implica que un fallo en un nodo no afectará a toda la red, sino que sólo afectará a dicho nodo y a sus consumidores. En cuanto a su identificación, cada nodo tiene asignado un nombre único que lo distingue de cualquier otro nodo de la red, evitando así conflictos de nombres. Por último, para la creación de dichos nodos, ROS proporciona librerías en diferentes lenguajes de programación que permiten crear e iniciar nodos en la red de ROS de una forma sencilla, siendo las más utilizadas la librería *roscpp* (C++) y la librería *rospy* (Python).

Debido a la abstracción que hace ROS, se utiliza una representación gráfica de cada uno de sus elementos, simplificando el despliegue de modelos. En dicha representación se usa el círculo como figura de representación del nodo, con el nombre de dicho nodo en el centro como se puede apreciar en la figura 5.1.

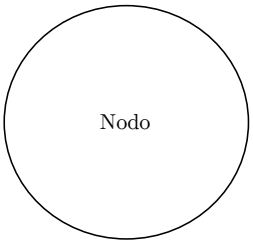


Figura 5.1: Representación gráfica de un nodo

Mensajes

Los nodos se comunican entre si a traves de mensajes que contienen datos con información relevante para ser intercambiada entre los nodos. Por defecto ROS tiene predefinidos un conjunto de mensajes primitivos que quedan recogidos en la tabla 5.1.

Tabla 5.1: Tipos de mensajes estándart definidos en ROS

Primitiva	Serialización	C++	Python
bool (1)	Entero de 8 bits sin signo	uint8_t (2)	bool
int8	Entero de 8 bits con signo	int8_t	int
uint8	Entero de 8 bits sin signo	uint8_t	int (3)
int16	Entero de 16 bits con signo	int16_t	int
uint16	Entero de 16 bits sin signo	uint16_t	int
int32	Entero de 32 bits con signo	int32_t	int
uint32	Entero de 32 bits sin signo	uint32_t	int
int64	Entero de 64 bits con signo	uint64_t	long
uint64	Entero de 64 bits sin signo	uint64_t	long
float32	Coma flotante de 32 bits	float	float
float64	Coma flotante de 64 bits	double	float
string	Ascii string (4)	std::string	string
time	Enteros de 32 bits con signo para seg/ nseg	ros::Time	rospy.Time
duration	Enteros de 32 bits con signo para seg/ nseg	ros::Duration	rospy.Duration

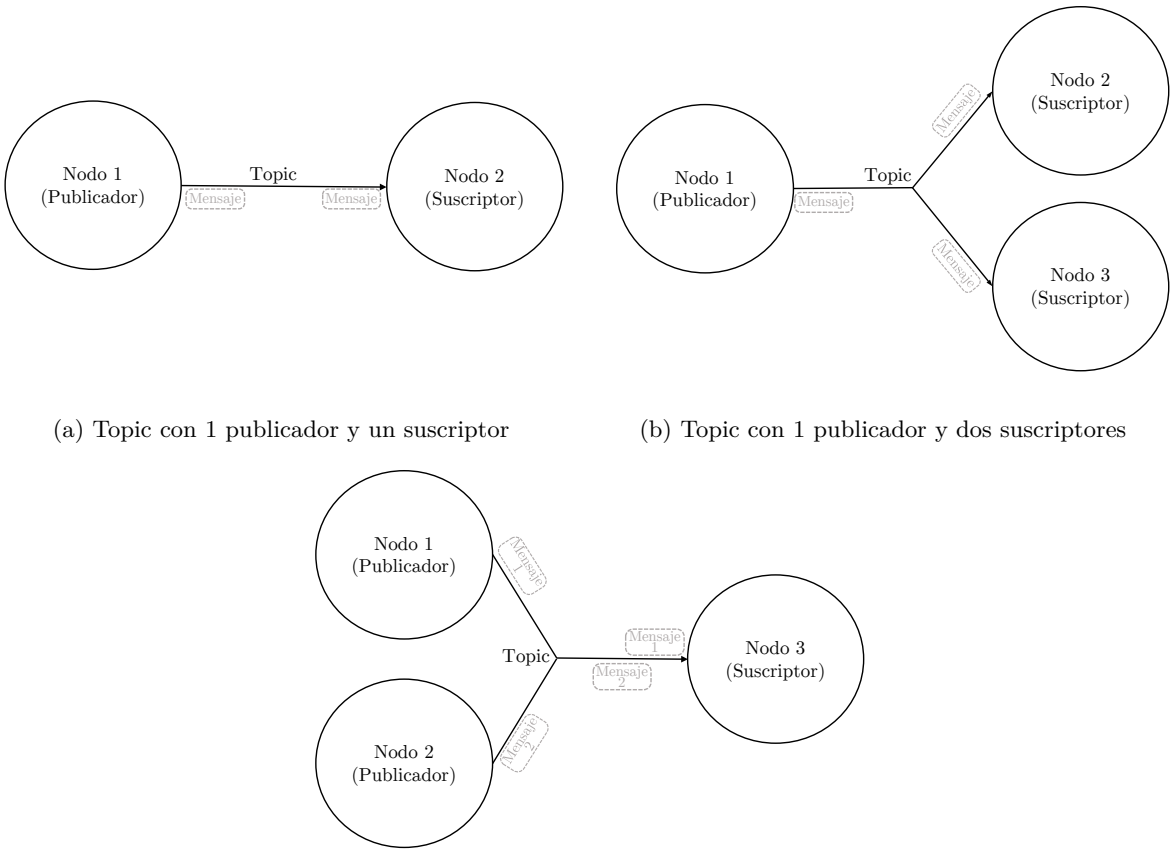
A parte de los tipos predefinidos, ROS permite crear nuevos tipos de mensajes a medida, haciendo uso de los tipos primitivos. Para definir un nuevo tipo de mensaje es necesario crear un archivo con extensión `.msg` en el que se define la estructura de datos del nuevo mensaje. A modo de ejemplo se define un mensaje para transportar información relativa a la velocidad angular y lineal de un robot:

```
1 # Mensaje para definir la velocidad angular
2 float32 vel_angular
3 float32 vel_lineal
```

El mensaje del ejemplo estará compuesto por dos estructuras de datos de tipo float con 32 bits de ancho de palabra, una para la velocidad angular y otra para la velocidad lineal. Para añadir más opciones, ROS permite crear vectores de tipos primitivos o mensajes propios, ampliando el abanico de posibilidades del transporte de información.

Topic

Los topics son buses de comunicación que transmiten mensajes entre nodos. Dicha transmisión se realiza sin una conexión directa entre nodos, lo que significa que productor y consumidor están desacoplados. Asímismo un topic puede tener varios nodos publicando en él y varios nodos suscritos a él. Hay que tener cuidado si existen varios nodos publicando en el mismo topic, ya que se pueden sobreescribir datos si dos o más nodos publican a la vez, no ocurriendo lo mismo en el caso contrario, puesto que la información que es publicada por un nodo le llega a la vez a varios suscriptores. Al igual que para los nodos, se va a hacer una representación gráfica del concepto, quedando reflejado en la figura 5.2.



(a) Topic con 1 publicador y un suscriptor

(b) Topic con 1 publicador y dos suscriptores

(c) Topic con dos publicadores y un suscriptor

Figura 5.2: Representación gráfica de un topic con sus 3 posibles opciones

En la figura 5.2a se pueden ver dos nodos conectados mediante un topic. Cuando el nodo emisor quiere transmitir un mensaje, pone dicho mensaje en el topic, el cual se encarga de transportar el mensaje hasta el nodo destino. En la figura 5.2b hay un nodo emisor y dos nodos consumidores. El nodo emisor opera igual que en el caso anterior, poniendo el mensaje en el topic para que éste transporte dicho mensaje hasta los dos nodos receptores. El tercer caso es el más problemático (figura 5.2c) ya que un mismo topic tiene dos nodos emisores y un solo receptor. Cada nodo envía un mensaje por el topic que le llega al receptor, suponiendo que los nodos no transmiten a la vez. Si transmitiesen a la vez el mensaje se sobreescribiría y al nodo receptor le llegarían datos corruptos. Los mensajes que se transmiten en el ejemplo pueden ser de todo tipo, desde cualquiera de los definidos por ROS hasta el mensaje de ejemplo que se expuso en la sección anterior.

Por defecto los topics de ROS transmiten haciendo uso del protocolo TCP/IP utilizando persistencia en las conexiones, conociéndose a este procolo de transporte como *TCPROS*. También es posible configurar los topics para que utilicen el protocolo de transporte UDP, el cual posee una baja latencia pero puede tener pérdida de paquetes. A este último protocolo de transporte se le conoce como *UDPROS*.

Servicios

El envío de datos a través de un topic se realiza mediante una difusión del mensaje publicado por el nodo origen en el topic hacia los nodos receptores suscritos a dicho topic. Este método de comunicaciones no permite realizar comunicaciones basadas en estructuras del tipo cliente-servidor con una comunicación bidireccional basada en peticiones y respuestas. Para solucionar este problema, ROS proporciona un

tipo de comunicación especial denominada servicios. Los servicios son desarrollados por el programador cuando los necesita, no existiendo servicios primitivos definidos por ROS como ocurría en el caso de los mensajes. Para crear un servicio es necesario definir la estructura de datos que se va a intercambiar entre los nodos cliente y servidor. Esta estructura se define de forma similar a los mensajes, creando un archivo con extensión `.srv` en el que se indica la estructura de datos de la petición del cliente y la estructura de datos de la respuesta del servidor. Para comprender mejor como funciona este tipo de estructura, a continuación se define un servicio que va a realizar las funciones de calculadora:

```
# Servicio para hacer una calculadora
float32 Operando_A
float32 Operando_B
string Operacion
---
float64 Resultado
```

Como se puede ver en el fragmento de código, el mesaje de petición está compuesto por dos operandos de tipo `float32` y el tipo de operación que viene determinado por un `string`. Por su parte, el servidor devolverá el resultado de la operación en un `float64`. En cuanto a la representación gráfica del servicio, se va a aprovechar el ejemplo de la calculadora para entender mejor como funciona este concepto básico de ROS.

La figura 5.3 ilustra como funcionaría una calculadora haciendo uso de un servicio en ROS. Existen dos nodos, un cliente que generará peticiones de operaciones, y un servidor que atenderá dichas solicitudes. Este último realizará las operaciones aritméticas requeridas para devolver posteriormente el resultado de la operación. En este ejemplo el nodo cliente inicia una solicitud de cálculo al servidor enviando en un mensaje de petición los dos operandos y la operación solicitada. Dicho mensaje se propaga hasta llegar al nodo servidor, el cual procesa la petición y realiza el cálculo correspondiente. Cuando ha realizado la operación el servidor envía un mensaje de respuesta hacia el cliente con el resultado de la operación solicitada. De esta forma ROS resuelve la carencia que presentan los topics ante escenarios en los que es necesario una arquitectura del tipo cliente-servidor.

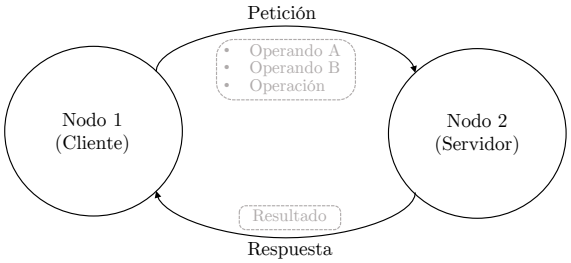


Figura 5.3: Representación gráfica del servicio calculadora

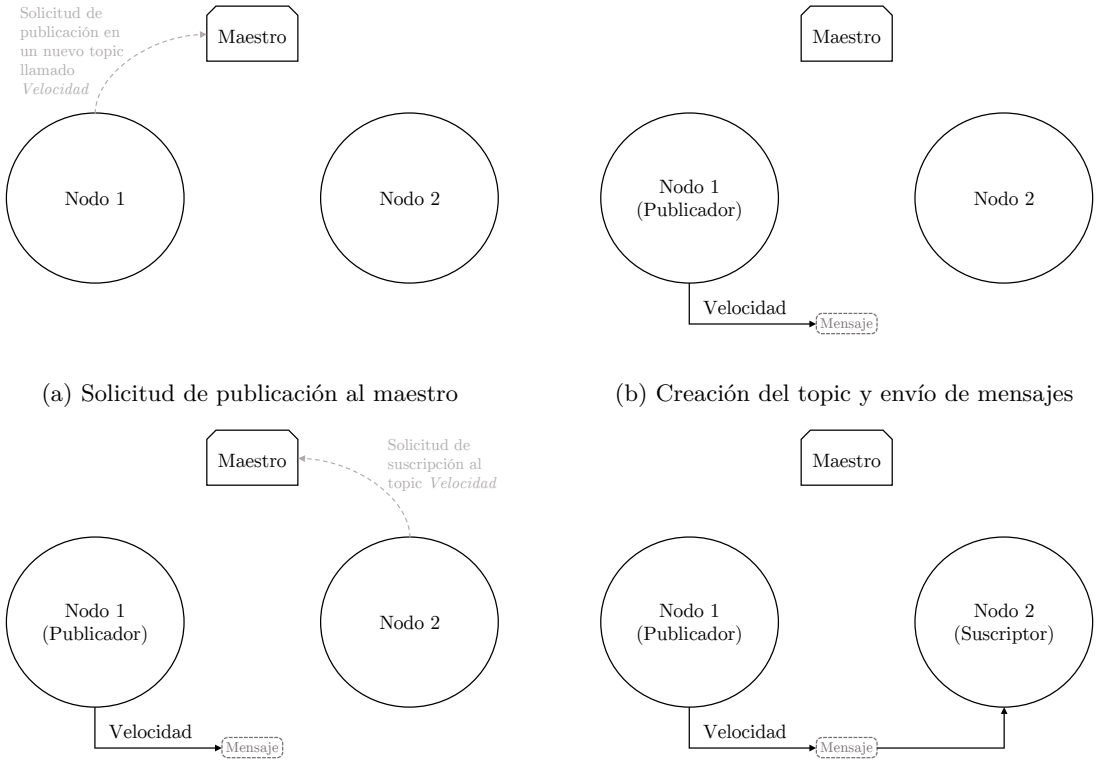
Servidor de parámetros

El servidor de parámetros es un diccionario que asocia nombres con valores por defecto y que puede ser accedido en toda la red de ROS. Los nodos utilizan este servidor de parámetros para guardar y proporcionar parámetros en tiempo de ejecución, aunque su diseño está pensado para guardar datos estáticos es habitual su uso para almacenar datos y fijar parámetros de configuración del robot.

Maestro

El maestro es el elemento que gestiona y controla la red de ROS. El maestro proporciona un registro de los nodos que hay en ejecución permitiendo la comunicación entre nodos. En sistemas distribuidos, el maestro debe estar en un equipo pudiendo ejecutar nodos en el mismo equipo y en otros equipos.

Para comprender mejor este concepto se va a ilustrar una solitud de trasvase de información entre dos nodos de ROS a través de un topic. En la figura 5.4 coexisten dos nodos y el maestro. En primer lugar (figura 5.4a) el nodo 1 inicia una solicitud de publicación en un nuevo topic llamado *Velocidad*. Cuando el maestro acepta la solicitud le proporciona los recursos necesarios al nodo para que cree el topic deseado y comience a transmitir datos por él (figura 5.4b). En este momento el nodo 1 se encuentra publicando en el topic, pero no hay ningún otro nodo a la escucha, por lo que los mensajes se pierden. Posteriormente el nodo 2 le indica al maestro que quiere suscribirse al topic *Velocidad*, el maestro gestiona todos los recursos y le indica a que topic debe conectarse (figura 5.4c). Finalmente el nodo 2 se engancha al topic y comienza a recibir mensajes enviados desde el nodo 1 (figura 5.4d).



(a) Solicitud de publicación al maestro

(b) Creación del topic y envío de mensajes

(c) Solicitud de suscripción al maestro

(d) Sucripción y recepción de mensajes

Figura 5.4: Representación gráfica de funcionamiento del maestro

5.2.2 Sistema de ficheros

Al igual que en un sistema operativo, un programa de ROS está dividido en carpetas que contienen subcarpetas y archivos con funcionalidades específicas, y cuya estructura queda recogida en la figura 5.5.

Como se puede apreciar, el sistema de ficheros de ROS está compuesto por dos componentes básicos, el metapaquete y el paquete. El sistema de ficheros contiene varios metapaquetes que a su vez cada uno contiene varios paquetes. Dichos componentes se van a analizar con mayor grado de detalle en las secciones posteriores.

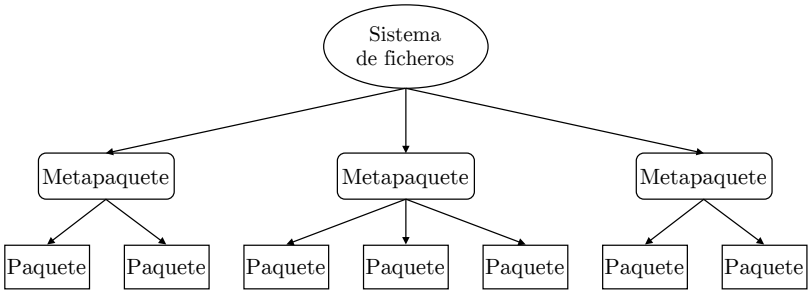


Figura 5.5: Estructura del sistema de ficheros de ROS

Paquete

El paquete es el componenente básico de ROS que contiene la estructura y el contenido mínimo para crear un programa en ROS. Además, un paquete permite agrupar un conjunto de nodos de ROS, servicios y/o bibliotecas, facilitando así su portabilidad. La estructura que presenta un paquete típico de ROS queda representado en la figura 5.6.

A continuación se va a explicar con más detalle cada uno de los elementos que componen un paquete de ROS:

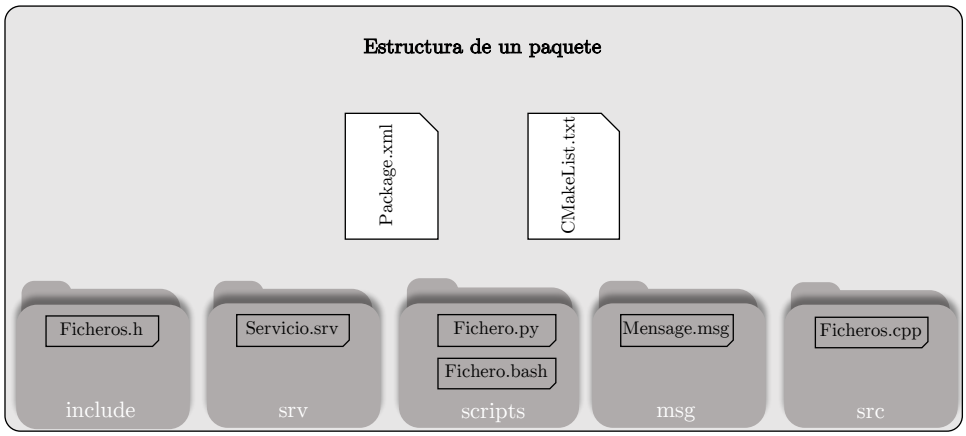


Figura 5.6: Representación gráfica de un paquete de ROS

- **Package.xml**: fichero que contiene información sobre el paquete, licencias, dependencias y *flags* de compilación.
- **CMakeList.txt**: fichero de entrada para el sistema de compilación de ROS necesario en la construcción de paquetes. En este fichero se indica el nombre del ejecutable, la ubicación de los archivos de código fuente del paquete, así como la creación de mensajes o servicios propios, e incluso el lugar en el que se desea instalar el paquete. Al crear el paquete el fichero viene comentado por completo, por lo que para generear los ejecutables es necesario descomentar líneas de código y añadir los campos deseados.
- **Directorio include**: carpeta donde se incluyen los archivos de cabecera de los programas. Si no hay archivos de cabecera esta carpeta permanecerá vacía.

- **Directorio srv**: carpeta en la que se incluye la descripción de las estructuras de datos de los servicios. En ella habrá tantos archivos con extensión `.srv` como servicios existan en el paquete. Si no existe ningún servicio esta carpeta no existirá.

- **Directorio msg**: carpeta en la que se incluye la descripción de las estructuras de datos de los mensajes propios. En ella habrá tantos archivos con extensión `.msg` como mensajes propios existan en el paquete. Si no existe ningún mensaje propio esta carpeta no existirá.

- **Directorio *scripts***: carpeta que contiene cualquier *script* de ejecución, ya sea un *script* de Python, bash, etc., Si no existe ningun script en el paquete esta carpeta no existirá.

- **Directorio src**: carpeta que contiene los archivos de código fuente del paquete. En ella se encontrarán archivos con extensión `.c` o `.cpp`. Si el paquete no contiene ningún paquete, la carpeta estará vacía.

Metapaquete

Los metapaquetes son elementos especiales que agrupan varios paquetes con el objetivo proporcionar una funcionalidad específica. Un ejemplo de ello es el stack de navegación orientado a la creación de mapas y navegación con Simultaneous Localization And Mapping (SLAM). Al contrario que ocurre con los paquetes básicos, los metapaquetes no contienen varios archivos y subcarpetas, sino que únicamente tienen en su interior un archivo llamado `package.xml` que incluye el sistema de compilación, las dependencias y un flag indicando que es un metapaquete.

5.2.3 Comunidad de soporte

La comunidad de ROS contiene recursos habilitados para la comunidad con el objetivo de intercambiar de archivos y conocimiento. Tiene varias áreas, entre las que se encuentran:

- **Distribuciones**: cada distribución de ROS es un conjunto de metapaquetes versionados que se pueden instalar, jugando un papel similar a las distribuciones de Linux. La distribución utilizada en este proyecto es ROS Indigo, que trabaja junto al sistema operativo Ubuntu 14.04. Esta versión era la última cuando comenzó el desarrollo del proyecto y tiene los mismos años de soporte que la versión de Ubuntu asociada (hasta abril de 2019). Sin embargo, ROS evoluciona constantemente, aparenciando nuevas versiones del sistema operativo robótico, siendo la última Kinetic Kame, asociada a Ubuntu 16.04 y con soporte hasta mayo de 2021.

- **Repositorios**: ROS tiene una serie de servidores réplica donde ubica el código de las distintas distribuciones. Además la institución que lo desee puede desarrollar y liberar su código para que sea accesible por toda la comunidad.

- **La Wiki**: es el principal foco de documentación que posee ROS. En ella se pueden encontrar tutoriales, ejemplos de código, manuales, aplicaciones, etc. Cualquiera puede crearse una cuenta y contribuir con su propia documentación, realizar correcciones, crear tutoriales, etc.

- **Listas de correo**: las listas de correo son el principal canal de comunicación para conocer las nuevas actualizaciones de ROS, así como un foro de soporte para resolver cualquier duda.

5.3 Estructura lógica de la capa funcional

Este apartado explicará el desarrollo *software* de la capa funcional, detallando cada uno de los bloques que la componen. Esta capa es la que tiene un contacto directo con el robot, por lo que su diseño debe ir en concordancia con sus capacidades físicas. Además, la inclusión de esta capa proporciona un nivel de abstracción a capas superiores, las cuales no deben preocuparse por como operar con el *hardware* que hay debajo.

La capa funcional está compuesta por dos bloques básicos: un primer bloque denominado módulo de comunicación, el cual lleva todo el control robótico, y un segundo bloque denominado modulo *safety*, encargado de detener el robot en el caso de que las capas superiores detecten algún problema. Cada bloque lleva integrado un servidor RPC que comunica la capa funcional con el ejecutor, de esta forma las capas superiores pueden estar separadas de la capa funcional de una forma física (en equipos distintos).

En la figura 5.7 se pueden ver los dos bloques mencionados anteriormente, el módulo de comunicación y el módulo *safety*, los cuales se comunican con el ejecutor a través de dos servidores RPC. Asímismo dentro cada uno de los dos bloques elementales coexisten varios módulos básicos que realizan una función específica. Cuando el ejecutor quiere ejecutar una orden, manda dicha orden al servidor RPC específico, el cual se encarga de traducir la orden y redirigirla al bloque correspondiente. Cuando el bloque termina la acción devuelve el resultado al servidor RPC, y éste a su vez lo reenvía hacia el ejecutor. Para conocer con mayor nivel de detalle la capa funcional se van a desarrollar por separado cada uno de los módulos que componen esta capa.

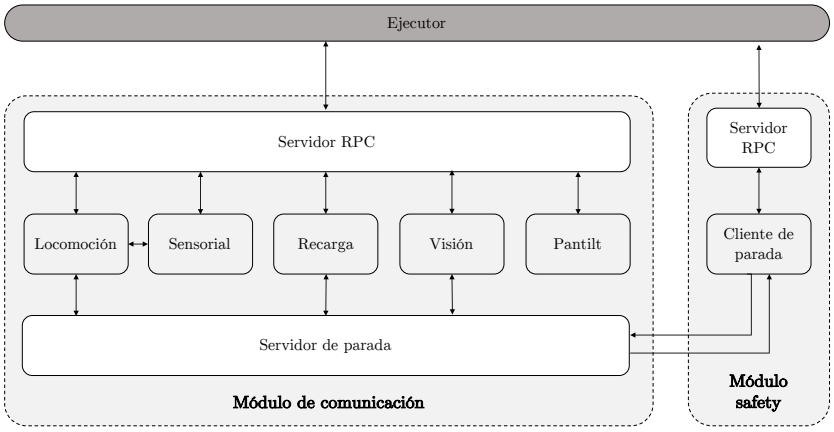


Figura 5.7: Diagrama de bloques de la capa funcional

5.4 Módulo de comunicación

Este módulo controla y gestiona todos los movimientos del robot (traslación, rotación, y posicionamiento de la *pantilt*) así como otras funciones básicas (captura de imágenes, carga automática y sensores). Dicho módulo se encuentra desarrollado dentro del *framework* de desarrollo robótico ROS, implementado y estructurado como un paquete de ROS. Además, para la creación de este paquete han sido necesarias dependencias de otros paquetes de ROS que contenían información y librerías con mensajes, aplicaciones y servicios específicos. Todas las dependencias utilizadas quedan recogidas en la tabla 5.2.

Tabla 5.2: Dependencias utilizadas para el módulo de comunicación

Depencia	Función
camera_calibration_parsers	Librería para escribir y leer parámetros de la Kinect
cv_bridge	Librería de interconexión entre ROS y OpenCV
geometry_msgs	Mensajes específicos para puntos, vectores y posiciones
image_transport	Librería para escribir y leer imágenes en los topic
roscpp	Librería en C++ con las funciones básicas de ROS
std_msgs	Mensajes estándart de ROS
kobuki_msgs	Mensajes específicos de la base Kobuki
kobuki_auto_docking	Librería para la carga automática
action_lib	Librería paara implementar un servidor especial de ROS
action_lib_msg	Mensajes específicos de un servidor especial de ROS

Como se puede ver en la tabla 5.2, se han utilizado librerías de visión para poder capturar imágenes con la Kinect, mensajes específicos de la base Kobuki para poder leer sus sensores, librerías específicas para realizar la carga automática, etc. Todas estas librerías siguen la fisolofía de ROS de reutilización de código ya que muchas de ellas, además de ser útiles para el TurtleBot II, también lo son para otros robots.

En cuanto a la representación gráfica de nodos, topics y servicios que se hacía en la sección 5.2, este módulo está compuesto por un solo nodo que atiende peticiones externas, suscribiéndose o publicando en diferentes topics y servicios en función de la petición que le llegue.

A continuación se van a analizar uno a uno los elementos que componen el módulo de comunicación.

5.4.1 Servidor RPC

RPC es un protocolo de comunicaciones que permite ejecutar código en otra máquina remota mediante la invocación de métodos o funciones y obtener resultados sin la necesidad de realizar conexiones basadas en sockets. En la actualidad existen varios estándares RPC entre los que destacan: DCE/RPC de *Open Software Fundation*, Distributed Component Model Object (DCOM) de *Microsoft* y Open Network Computer Remote Procedure Call (ONC RPC) de *Sun* (RFC 1057) con su programa RPCGEN. Ésta ultima implementación de RPC ha sido la que se ha utilizado en el proyecto para implementar los dos servidores RPC. Para generar el servidor RPC con la herramienta RPCGEN únicamente hay que definir en un arhivo de configuración las estructuras de datos que se van a utilizar y las funciones que van a ser llamadas a través de los procedimientos remotos. Cuando todas las estructuras y funciones estén definidas, la herramienta RPC generará el esqueleto del código fuente que implementará la funcionalidad, así como el soporte cliente-servidor. Para generar todo este código solo es necesario ejecutar el siguiente comando en una terminal:

```
1 rpcgen -a <mi_archivo_configuracion.x>
```

La ejecución de esta orden crea varios archivos con código fuente, de los cuáles se identifican dos partes bien definidas: los archivos del cliente y los archivos del servidor. Únicamente hay que implementar la parte del servidor en la que se realizan llamadas a las funciones previamente definidas en el archivo de

configuración. Estas funciones están vacías cuando la herramienta genera el código fuente, por lo que es necesario terminar su implementación.

5.4.2 Bloque de locomoción

El bloque de locomoción será el encargado de mover el robot a través de sus dos ruedas motrices teniendo en cuenta el estado de los sensores del robot para poder detenerlo caso de colisión. Este bloque está compuesto por dos funciones básicas que se comunican con el servidor RPC: avance y giro. Dichas funcionees reciben como parámetros de entrada la distancia o giro a recorrer y las velocidades máximas (angular y lineal) que el robot alcanzará. Además, dichas funciones hacen uso de un control Proporcional Integral (PI) para que el robot reduzca la velocidad al llegar a su objetivo, por lo que en primer lugar se va a explicar como funciona un control PI.

Un controlador PI es un mecanismo de control por realimentación de bucle cerrado comunmente extendido en sistemas de control industrial que tiene como objetivo minimizar el error a la salida de un sistema. Para conseguirlo, calcula el error entre un valor medido y un valor desado y realimenta la entrada del sistema con dicha medida. En base a ese error el algortimo de control ajusta los parámetros de entrada para que el error final del sistema sea cero. En cuanto a su composición, los algoritmos de control PI constan de dos miembros: la acción de control proporcional (P) que obtiene una salida proporcional al error, y la acción de control integral (I) que consigue una salida proporcional al error acumulado, quedándo recogida en la ecuación 5.1 la expresión que define este mecanismo de control.

$y(t)$ se corresponde con la salida del controlador, K_p es la constante de acción proporcional y K_i es la constante de acción integral. El resultado de la acción de control proporcional se consigue multiplicando la constante de acción proporcional por el error calculado, mientras que el resultado de la acción de control integral viene determinado por la multiplicación de la constante de acción integral por la integral del error en un periodo de tiempo. Para obtener el resultado final es necesario sumar los resultados obtenidos en las operaciones anteriores.

$$y(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(\tau) \cdot d\tau \tag{5.1}$$

El controlador PI que se va a implementar en el bloque de locomoción actuará sobre la velocidad del robot (salida del controlador) en función de la distancia que queda por recorrer (error del sistema), consiguiendo así un control de velocidad progresivo en el que la velocidad del robot es proporcional a la distancia que le queda por recorrer. Además, al efectuar los movimientos de parada con velocidades lentas se evitan problemas de patinaje en las ruedas del robot, lo que provocaría errores en la odometría.

Para realizar el ajuste de los parámetros del PI se ha hecho uso del método Ziegler-Nichols [34], publicado en el año 1942 y que permite realizar el ajuste del controlador PI de forma empírica sin la necesidad de conocer las ecuaciones del sistema a controlar. Este método permite fijar las ganancias de acción proporcional y acción integral teniendo en cuenta la oscilación del sistema, y cuyo método de ajuste se detalla a continuación:

1. Se utiliza únicamente el control proporcional fijando su constante K_p en un valor bajo. Progresivamente se incrementa la constante hasta que la salida del sistema comience a oscilar. En ese instante se anota el valor de la constante (valor crítico) y el periodo de oscilación (K_c y P_{osc})

2. Teniendo en cuenta el valor crítico y el periodo de oscilación se aplican las ecuaciones 5.2 y 5.3:

$$K_p = 0{,}45 \cdot Kc \tag{5.2}$$

$$K_i = \frac{K_p \cdot 1{,}2}{P_{osc}} \tag{5.3}$$

3. Se hace un ajuste fino de forma manual en los valores de las constantes para que el sistema tenga el comportamiento deseado.

Una vez que se ha entendido el mecánismo básico de control del que hacen uso las funciones de avance y giro, se procede a explicar cada una de ellas.

Avance: como se ha indicado anteriormente, como parámetros de entrada recibe las velocidades máximas que alcanzará el robot y la distancia que debe recorrer. Como resultado devuelve una estructura de datos que contiene la posición del robot al finalizar el movimiento descompuesta en en coordenadas x e y, junto con la orientación del robot y una variable de error que indica si el movimiento se ha realizado correctamente o ha ocurrido algún error durante su ejecución. La función, gracias al algoritmo PI que tiene implementado, ajusta la velocidad del robot en función de la distancia que le queda por recorrer, decrementándola conforme llega a su objetivo. Además, esta función recibe información desde el bloque sensorial para detener el robot en el caso de encontrar algún obstáculo. Para llevar a cabo estas funciones ha sido necesario suscribirse y publicar en dos topics de la red de ROS, uno para obtener los datos de odometría del robot y a partir de ellos calcular la distancia recorrida, y otro en el que se publica la velocidad que deben llevar los motores del robot. El topic `/odom` es el encargado de transportar la odometría del robot en mensajes del tipo `nav_msgs/Odometry`, los cuales contienen una cabecera y un número de secuencia junto a la posición del robot en forma de cuaterniones [*x, y, z, w*] dónde x e y son las posiciones del robot en el plano y z y w proporcionan la orientación del robot. La medida odométrica recogida por este topic se calcula con respecto a la posición y orientación del robot en el momento que arranca la red de ROS. En cuanto al ajuste de la velocidad del robot, ésta se hace a través del topic `/cmd_vel_mux/input/teleop` que transporta mensajes del tipo `geometry_msgs/Twist` en los que se incluye la velocidad lineal y angular. El modo de operar de la función es el siguiente: se recibe información del topic de odometría para calcular la distancia que ha recorrido el robot a partir del cuaternio de posición y se le descuenta esa distancia a la distancia total que debe recorrer, obteniendo el error del algoritmo PI. Con ese error se alimenta el controlador PI, el cual calcula la velocidad lineal del robot y la publica en el topic de velocidad. El proceso se repite periódicamente hasta que el robot alcance su objetivo. En caso de que el bloque sensorial envíe una señal de alarma el proceso se interrumpe, deteniendo el robot inmediatamente para evitar posibles colisiones. Al finalizar la acción de avance se recoge la posición del robot así como la variable de error para devolver los datos al ejecutor a través del servidor RPC.

Giro: al igual que la función anterior, recibe el ángulo de giro deseado y la velocidad angular máxima que alcanzará el robot, devolviendo la misma estructura de datos que devolvía la función de avance. La lógica de esta función es similar a la de la función de avance, suscribiéndose y publicando en los mismos topics, solo que en vez de avanzar, ahora el robot girará sobre sí mismo. En cuanto a la secuencia de operaciones, los pasos son similares a los de la función anterior: se recoge la orientación actual del robot en el topic de odometría y se calcula el ángulo de giro que queda para llegar al destino, obteniendo los grados de giro que quedan por recorrer. Este dato se le introduce al controlador PI para que ajuste consecuentemente la velocidad angular del robot a través del topic de velocidad, repitiéndose el proceso hasta que se alcance el objetivo final o hasta que el bloque sensorial envíe una señal de alarma. Cuando acaba el proceso iterativo se recoge la posición del robot junto con la variable de error para reenviar toda la información al ejecutor a través del servidor RPC.

5.4.3 Bloque sensorial

El bloque sensorial es el encargado de leer los sensores que posee el robot, y cuyo cometido, es presevar la integridad del robot detectando objetos que puedan aparecer y monitorizando el estado de la batería del robot. Este bloque está compuesto por 3 componentes encargados de comprobar los sensores del parachoques, los sensores sonar y el nivel de carga de la batería, así como si el robot se encuentra cargando o descargándose. A continuación se va a detallar cada una de los 3 componentes:

Sensores parachoques: este componente monitoriza el estado del parachoques del robot comprobando si algún elemento ha colisionado con él. Para transferir la información del estado del parachoques al ejecutor se ha generado una función que conecta el servidor RPC con ROS. Dicha función no tiene parámeros de entrada y como resultado devuelve un entero con el número de parachoque o parachoques que colisionaron, dónde el 1 representa el sensor derecho, el 2 el sensor central y el 3 el sensor izquierdo. Si varios sensores permacenen pulsados el valor devuelto será la suma de sus pesos, o si por el contrario no ha colisionado ninguno, devuelve cero. La función obtiene el estado del parachoques suscribiéndose al topic que proporciona la base Kobuki `/mobile_base/events/bumper`, en el cual se publica el estado del parachoques (pulsado o no) y el número de pulsador que colisionó. Toda esta información se transmite en mensajes del tipo `/kobuki_msgs/BumperEvent` que son procesados para poder devolver al ejecutor la información que necesita.

El estado del parachoques también es accedido desde el módulo de locomoción con el objetivo de detener el robot en caso de colisión. Para ello desde dicho bloque se ha hecho una suscripción al topic `/mobile_base/events/bumper`. Los datos recibidos por el topic son analizados y en el caso de detectar una colisión se activa un flag de parada que detiene el bloque de locomoción.

Sensores Sonar: la inclusión de los sensores sonar en el robot se hizo en una segunda iteracción después de comprobar que la detección de obstáculos por alcance era un método muy precario. Para añadir este tipo de sensores se incorporó un Arduino junto a los sensores sonar vistos en el apartado 3.5.3, integrando todos los elementos en ROS con ayuda de la librería *rosserial*. Esta librería permite crear topics en el Arduino transmitiendo los datos de dichos topics a la red de ROS mediante una conexión USB con el ordenador. De esta forma, el cálculo de la distancia proporcionado por los sensores ultrasónicos se transmite directamente desde el Arduino a dos topics (uno por sensor) cuyos nombres son `/sonar1` y `/sonar2`. Estos topic transportan en mensajes de tipo `/std_msgs/Int32` la distancia a los obstáculos medida en milímetros con una frecuencia de actualización de 100 ms (tiempo suficiente para detectar un obstáculo cuando el robot está en movimiento).

La información de los sensores sonar únicamente se recoge desde el bloque de locomoción para averiguar si existe algún objeto que interfiera en la trayectoria del robot. Desde el ejecutor no es necesario tener constancia de este sensor ya que el propio bloque de locomoción detiene el robot en el caso de que algún obstáculo se aproxime de forma inminente, devolviendo un flag de error que indica que el robot se detuvo por un obstáculo. Por lo tanto el bloque sonar se suscribe a los topics `/sonar1` y `/sonar2` generados por el Arduino. Para no tener que estar analizando constantemente la distancia a los objetos se ha incluido un procesado de la información consistente en un filtro y el establecimiento de un valor umbral. Este procesado consiste en un filtro que calcula la media de las 3 últimas muestras recogidas por el sensor de sonar con el objetivo de eliminar medidas erróneas. Además se ha fijado un valor umbral de 30 cm a partir del cual, si la media del filtro es menor, se activa un flag de parada que recibe el bloque de locomoción para detener el robot.

Estado de la batería: con este componente se pretende que el ejecutor conozca el nivel de batería del robot así como si se encuentra conectado a la red eléctrica o está en estado de descarga. Para conseguir este objetivo se han creado dos funciones que conectan el servidor RPC con ROS: la primera de ellas extrae el nivel de batería y la segunda comprueba si el robot se encuentra en la base de carga. Ambas funciones no tienen ningún parámetro de entrada y como resultado la primera devuelve un número real con el nivel de batería y la segunda un entero indicando si está conectado al cargador o no. Toda esta información se recoge de del topic `/diagnostics` cuyos datos viajan en mensajes del tipo `/diagnostic_msgs/DiagnosticArray`, un vector con varios campos que indican el estado del robot, entre los que se encuentran el nivel de batería y el estado de carga (cargando o descargando). Esta información va contenida en variables de tipo `string`, por lo que es necesaria su transformación para adecuarla a los valores que devuelven las funciones de conexión con el servidor RPC.

5.4.4 Bloque de recarga

La carga de las baterías es un elemento crítico para un robot autónomo, ya que si éstas se descargan el robot no puede moverse por sí mismo. Por este motivo se ha incorporado un módulo de recarga que añade nuevas funcionalidades, permitiendo que el robot se dirija a la estación de carga de forma autónoma. Para ello, este bloque hace uso del paquete `/kobuki_auto_docking` que trae implementado ROS. Este paquete hace uso de los LED infrarrojos que tiene la base de carga, así como los receptores infrarrojos que incluye la base Kobuki en su frontal. La base de carga posee 3 emisores infrarrojos que dividen 3 regiones: izquierda, derecho y centro, las cuales se subdividen en otras dos áreas: zona cercana a la base y zona lejana a la base. De esta forma es fácil ubicar la zona en la que se encuentra el robot a la hora de realizar la recarga y por consiguiente reconducir el robot hacia la base de carga. La idea es simple, si el robot se encuentra en la zona central mirando de frente a la base, únicamente ha de dirigirse hacia la base siguiendo una trayectoria recta. Si por el contrario se encuentra en la región izquierda, el robot empezará a girar hasta encontrar la región izquierda con su sensor derecho. En ese momento el robot se encuentra ubicado perpendicularmente hacia la zona central, por lo que ha de avanzar hasta que el sensor derecho detecte la zona central. Cuando lo encuentra, el robot gira hasta que el sensor frontal detecte la zona central. Al encontrarlo el robot unicamente ha de seguir un movimiento de avance lineal para llegar a la base. Si el robot se encuentra en la zona derecha el proceso sería el mismo solo que invirtiendo los movimientos.

Este paquete lleva implementado un servidor que atiende peticiones para inicar o detener el proceso de búsqueda de la estación de carga a través del envío de mensajes especiales a dicho servidor. Para trabajar con el servidor de recarga se han creado una serie de funciones que a través del servidor RPC se comunican con el ejecutor, las cuales se describen a continuación.

Arranque: permite enviar la orden que inicia el proceso de búsqueda de la estación de carga. Para conseguirlo se ha implementado una función en el servidor RPC sin parámetros de entrada y que devuelve al ejecutor un entero indicando si la conexión a la base de carga ha sido satisfactoria, o por el contrario se ha producido algún error. Para enviarle la petición al servidor se ha hecho uso del mensaje `/konuki_msgs_AutodockingAction`, un mensaje especial compuesto por una variable en la que va la petición de arranque, una variable de feedback y una variable de resultados. Con este mensaje especial es posible obtener el resultado que devuelve el servidor, así como el estado de la petición que se le envía. El proceso que se sigue para enviar la petición es el siguiente: en primer lugar se inicia la conexión con el servidor, si no hay respuesta por parte del servidor se envía un mensaje de error al ejecutor. Si la conexión se realiza satisfactoriamente, se crea el mensaje especial y se envía hacia el servidor esperando

la confirmación durante un periodo de tiempo. Si no se recibe respuesta se envía un mensaje de error al ejecutor y si se recibe, se envía hacia el ejecutor un mensaje de confirmación.

Rearranque del servidor: depurando el bloque de arranque se observó que el servidor de recarga era inestable y sufría continuas caídas. Para resolverlo se creó este bloque que permite rearrancar dicho servidor en caso de caída. Ha sido necesario añadir una función nueva en el servidor RPC que no contiene parámetros de entrada, y como resultado devuelve un entero con el resultado de la operación. Esta función hace uso de la creación de procesos para rearrancar el servidor, creando un nuevo proceso hijo en el que se ejecuta la orden de arranque del servidor. Por su parte, el proceso padre se encarga de verificar que el servidor ha arrancado correctamente devolviendo el resultado al ejecutor a través del servidor RPC.

5.4.5 Bloque de visión

El bloque de visión se encarga de la parte de visión incorporada en el robot. Actualmente está destinado a la captura de fotografías así como al rearranque de los servicios de los que hace uso la cámara si se caen. Se han creado dos componentes que realizan estas dos tareas: captura de imágenes y rearranque de la cámara.

Captura de imágenes: como su nombre indica, este componente se encarga de realizar la captura de imágenes a través de la cámara del robot. Al igual que en casos anteriores, se crea una función en el servidor RPC que comunique el ejecutor con ROS. Dicha función no contendrá parámetros de entrada y como resultado devolverá un entero indicando si ha sido posible hacer la fotografía, o por el contrario ha ocurrido algún error. La fotografía en sí no es enviada al servidor RPC, sino que se guarda en una carpeta dentro del ordenador que gobierna el robot. Esta fotografía puede ser accedida o transportada a otro ordenador haciendo uso del protocolo de comunicaciones ssh.

La camára está constantemente grabando y publicando cada fotograma en el topic `/camera/rgb/image_raw`, por lo tanto para realizar una fotografía basta con suscribirse a dicho topic y procesar el primer mensaje recibido. Dichos mensajes son del tipo `sensor_msgs/Image`, los cuales contienen una cabecera, la anchura y altura de la imagen, la codificación, organización de los datos, el tamaño total de la imagen y la imagen en sí. Además, para poder guardar la imagen en un formato legible (`.jpg`, `.png`, `.bmp`, etc.) es necesario transformar la información del mensaje `sensor_msgs/Image` a dicho formato haciendo uso de la librería *OpenCV* implementada en ROS. La funcion devuelve al servidor RPC un número negativo en caso de error o un número positivo si la captura se ha realizado con éxito.

Rearranque de la camara: a veces es posible que al arrancar el sistema al operador se le olvide poner en marcha los servicios que activan la cámara, por lo que se ha añadido una función que los arranca si por algún motivo no se han puesto en marcha o se han caído. El método aplicado es similar al del rearranque del servidor de recarga, implementando una nueva función en el servidor RPC que inicie los servicios de la cámara. La función no tiene parámetros de entrada y como resultado devuelve un entero indicando si la operación ha tenido éxito o no. Al igual que para el rearranque del servidor de recarga se ha creado un nuevo proceso hijo que activa los servicios, mientras que el proceso padre devuelve al ejecutor el resultado de la operación.

5.4.6 Bloque *pantilt*

La *pantilt* se encarga de reorientar la cámara incorporada en el robot para poder tomar fotografías en varias posiciones sin la necesidad de mover al robot. La comunicación de la *pantilt* con el ejecutor se realiza como en todos los ejemplos anteriores, a través del servidor RPC. La función que se ha incluido en dicho servidor para controlar la *pantilt* recibe como parámetros la posición de *yaw* y *pitch* que tendrá la *pantilt* y como resultado devuelve un booleano que indica si la operación se ha realizado con éxito o no. Como ya se vio en la sección 3.5.2, éste elemento está controlado por un Arduino Uno que mediante señales PWM controla la posición de dos servomotores. En cuanto a la comunicación con ROS por parte del Arduino, se realizado de la misma forma que en los sensores sonar, haciendo uso de la librería *rosserial*, permitiendo enviar la información de la pantilt al Arduino a través del topic `/pantilt`. Este topic transporta un mensaje del tipo `std_msgs/Int32MultiArray` que contiene dos enteros de 32 bits en los que se coloca el ángulo de *yaw* y *pitch* respectivamente. El modo de operación de la función es el siguiente: el servidor RPC recibe los datos de *yaw* y *pitch*, convirtiéndolo en un mensaje del tipo `std_msgs/Int32MultiArray` para posteriormente enviarlos por el topic `/pantilt`. Arduino recibe dicha información a través del topic y la transforma para generar la señales PWM que mueven los servomotores. Al terminar el movimiento Arduino devuelve el resultado al servidor RPC, que a su vez lo renenvía al ejecutor.

5.4.7 Servidor de parada

Algunas de las acciones que ejecuta la capa funcional deben ser detenidas por parte del ejecutor cuando detecta que algo no va bien. Por este motivo se ha incluido un servidor de parada que pueda atender dichas peticiones. La estructura que presenta la figura 5.7 es algo atípica, ya que el servidor de parada no atiende directamente las peticiones del servidor RPC, sino que utiliza un cliente propio para atender estas peticiones. Esto es debido a que el servidor de parada está implementado como un servicio de ROS en el que un nodo cliente envía las peticiones al servidor y éste actúa consecuentemente sobre los bloques del módulo de comunicación. De esta forma se consiguen diferenciar dos grandes bloques que añaden modularidad a la capa funcional.

En cuanto a la implementación del servidor, se ha realizado haciendo uso de los servicios de ROS, creando un servicio propio llamado `StopCommand`. Este servicio utiliza una estructura de mensaje propia llamada `/stopNode/stopcommandsrv`, que contiene un mensaje de petición y otro de respuesta. El mensaje de petición es un entero que indica que bloque se ha de detener (1 para el bloque de locomoción, 2 para el de visión y 3 para el de recarga), mientras que el mensaje de respuesta incluye un booleano que indica si la se ha detenido correctamente el bloque o no. Por lo tanto, cuando una petición llega a éste servidor es analizada para posteriormente detener el bloque específico. La detección de los bloques de locomoción y visión se realiza activando un flag que paraliza las operaciones en curso que tiene cada uno de los bloques. Sin embargo, la detección del bloque de recarga no se puede hacer de este modo debido su dependencia con un servidor externo, por lo que la detección de este bloque pasa por enviar un mensaje de cancelación al servidor ROS que contiene la lógica de la búsqueda de la estación de carga. Este mensaje se envía a través del topic `/dock_drive_action/cancel`, recibiéndose en el servidor y cuyo efecto es la detección de la búsqueda de la base de carga. Cuando finaliza la acción, el servidor de parada envía el resultado de la operación al cliente implementado dentro del módulo *safety*.

5.5 Modulo *safety*

Este módulo se encarga de que la capa funcional se mantenga dentro de los valores nominales de operación, pudiendo actuar sobre el módulo de comunicación en situaciones de peligro. A través de este módulo es posible detener el bloque de locomoción, el bloque de visión y el bloque de recarga desde el ejecutor, con la ayuda de un servidor RPC. El servidor RPC se ha implementado de forma similar al bloque anterior, haciendo uso de la herramienta RPCGEN. Al contrario que el módulo anterior, este bloque apenas tiene dependencias de otros paquetes ROS, ya que sólo hace uso de los mensajes estándart `std_msgs` y de la librería `roscpp`. Además en este módulo únicamente hay implementado un cliente de un servicio ROS, con lo que la simplicidad del módulo aumenta considerablemente.

La comunicación desde el ejecutor se hace a través del sevidor RPC, el cual tiene implementada una función a la que se le pasa como parámetro de entrada un carácter que identifica el bloque a detener, y como resultado devuelve un entero con el resultado de la operación. Cuando llega una orden desde el ejecutor, la función implementada en el servidor RPC traduce el carácter recibido y crea un cliente especial para el servicio (`StopCommand`), enviándole la petición de detección del bloque solicitado (locomoción, visión o recarga) en un mensaje del tipo `/stopNode/stopcommandsrv`. Cuando el servidor procesa y ejecuta la acción devuelve un resultado que es traducido por el cliente de parada y reenviado hacia el ejecutor a través del servidor RPC.

Capítulo 6

El ejecutor

Este capítulo se centrará en explicar la capa intermedia que conforma la arquitectura MoBAr. En primer lugar se hará una introducción al lenguaje programación y a la herramienta empleada para el desarrollo de esta capa. Despues se mostrarán los conceptos claves y el diseño del ejecutor. Finalmente se detallará el desarrollo *software*, dividiendo el capítulo en dos apartados: los adaptadores y el modelado del plan.

6.1 Introducción

El ejecutor de la arquitectura está implementado con Plan Execution Interchange Languaje (PLEXIL) [35], un lenguaje de modelado y representación de planes de ejecución que actualmente se encuentra en la versión 4.0, aunque en este proyecto se utiliza la versión 2.50. Mientras el Universal Executive (UE) es el sistema que se emplea para ejecutar los planes desarrollados en PLEXIL. Tanto PLEXIL como el UE han sido diseñados y desarrollados por el ARC y el JPL de la NASA junto con la Universidad Carnegie Mellon. Ambos componentes han sido intregrados en elementos reales para demostrar su potencial, por ejemplo en el rover K10, en pruebas de movilidad en superficie y en la interacción hombre-robot, así como en la automatización en ciertas tareas de la Estación Espacial Internacional.

El objetivo de PLEXIL y del UE es crear un lenguaje bien definido para expresar planes que controlen sistemas complejos. Además, el lenguaje PLEXIL permite representar planes muy complicados con sentencias de alto nivel como bucles, sentencias condiciones, secuencias ordenadas, secuencias en paralelo, etc., que abren el abanico de posibilidades a la hora de programar planes de ejecución. Otra característica que proporciona este lenguaje es la posibilidad de obtener datos de sistemas externos sin la necesidad de acceder a una base de datos de telemetría gracias a la incorporación de diferentes métodos de conexión con el exterior.

6.2 El lenguaje PLEXIL

Los planes generados en lenguaje PLEXIL están compuestos por una descomposición jerárquica de tareas que deben llevar a cabo sistemas externos. Estos planes pueden estar compuestos por uno o más archivos codificados en lenguaje XML , aunque la programación se realiza en lenguaje PLEXIL que un compillador se encarga de traducir a formato XML.

PLEXIL define un elemento básico sobre el que articulan todos los planes: el nodo. Seis son los tipos de nodos existentes en PLEXIL en función de la tarea a realizar:

- **Vacío:** nodo que no realiza ninguna acción, sirve como elemento bloqueante o de espera. En la práctica son poco comunes, y únicamente pueden tener atributos en su interior.
- **Asignación:** asigna valores a variables de un plan mediante una expresión lógica o matemática.
- **Comando:** tiene como objetivo la comunicación con elementos externos al ejecutor (en este proyecto las capas deliberativa y funcional). El funcionamiento de este nodo consiste en llamar a una función externa con los parámetros adecuados, permitiendo así la comunicación con elementos externos a PLEXIL. Además este tipo de nodos no bloquean la ejecución del plan, sino que se realiza de forma asíncrona, permitiendo que el plan de PLEXIL continúe con su ejecución a la vez que se ejecutan comandos externos. Esta comunicación con sistemas externos se realiza a través de interfaces que que realizan la comunicación entre sistemas, vistas con más detalle en la sección 6.4.
- **Actualización:** nodo que actualiza variables del plan con datos provenientes del mundo exterior.
- **Librería:** este nodo permite incluir un plan de PLEXIL dentro de otro plan, garantizando así la modularidad de los planes de ejecución. Gracias a este nodo es posible desarrollar planes sencillos reutilizables en planes complejos.
- **Lista:** nodo que contiene más nodos en su interior. Permite crear la jerarquía del plan de PLEXIL incluyendo en su interior a cualquier tipo de nodo (incluidos los nodos lista).

Asímismo cada nodo en PLEXIL solamente podrá contener 4 elementos en su interior:

- **Variables:** existen 4 tipos de variables (booleanos, enteros, reales y cadenas de caracteres), pudiendo permitir crear arrays de cada uno de los tipos de variables. Internamente PLEXIL trata a todas las variables como tipo double, permitiendo así una rápida conversión entre tipos en los *Interface Adapters* (ver sección 6.3.1).
- **Condiciones:** controlan el ciclo de ejecución del plan de PLEXIL. Estas condiciones vienen definidas por el propio plan de PLEXIL o por elementos del mundo externo, condicionando la ejecución de cada nodo. Existen dos tipos de condiciones: las *gate conditions* que comprenden las condiciones de inicio,finalización repetición y omisión, y las *check conditions* que contienen las precondiciones, postcondiciones y condiciones invariantes. Mientras que las *gate conditions* determinan el comienzo y fin de la ejecución del nodo, las *check conditions* establecen las condiciones para determinar si la ejecución del nodo fue correcta. Al elaborar los planes de PLEXIL las condiciones serán útiles para permitir la ejecución de un nodo en función del resultado de la ejecución de otro nodo.
- **Recursos:** PLEXIL permite gestionar el uso de recursos que utilizarán los nodos comando.
- **Cuerpo:** el cuerpo del nodo contiene la lógica que rige la acción realizada por el nodo y que condiciona el tipo de nodo. Es la parte más importante del nodo junto con las condiciones ya que en su conjunto definen el comportamiento de dicho nodo.

Para realizar la comunicación con sistemas externos PLEXIL proporciona dos métodos de comunicación, los *comandos* que permiten enviar tareas a la capa funcional mediante intercambio de parámetros, y los *lookups* que permiten leer el estado del mundo de forma asíncrona asignando el valor leído a una variable. Mientras que los *comandos* solo pueden ser definidos en el cuerpo del nodo, los *lookups* solo se pueden definir en las condiciones del nodo.

También es interesante conocer las herramientas que proporciona el *framework* de desarrollo de PLEXIL. La primera de ellas es PlexilScript junto con el programa TextExec, los cuales permiten simular el

comportamiento de los sitemas externos que interaccionan con PLEXIL. La segunda herramienta es la aplicación Lightweigth Universal Executive (Luv) que permite simular los planes de forma gráfica, viendo en todo momento el estado de los nodos. Además tiene la opción de poder ejecutar los planes paso a paso o poder fijar puntos de parada, lo que permite depurar los planes con un alto nivel de detalle.

6.3 El ejecutor UE

El UE es el sistema que se encarga de interpretar y ejecutar los planes de PLEXIL. El UE se puede ver como un *framework* pensado para cargar y ejecutar los planes escritos en PLEXIL dentro de nuestro programa de control. Asímismo el UE también es capaz de comunicarse a través de interfaces con sistemas de bajo nivel que interactúan con el mundo exterior. La ejecución del UE se lleva a cabo a través de ciclos de ejecución que detallan las variaciones de estado de los nodos en un instante de tiempo específico. En cuanto a la ejecución del plan, se hace en pasos discretos, procesando los eventos según el orden de llegada. De esta forma eventos externos pueden afectar directamente en la ejecución del plan, pudiéndose obtener resultados distintos tras ejecutar planes iguales.

El UE está compuesto por dos elementos: el UE propiamente dicho, encargado de leer, interpretar y ejecutar los planes de PLEXIL, y una o varias interfaces que comunican al ejecutor con sistemas externos. Los siguientes apartados explican el funcionamiento básico de estas interfaces, así como su configuración.

6.3.1 *Interface Adapters*

Para realizar la comunicación con sistemas externos, el UE necesita librerías dinámicas que le comuniquen con el mundo exterior. Estas librerías son conocidas por el nombre de *Interface Adapters* y se componen de llamadas a los miembros de la clase `InterfaceAdapter` programados en C++. Esta clase `InterfaceAdapter` proporciona el soporte necesario para cubrir nuestras necesidades, ya que incluye funciones virtuales que permiten codificar los *comandos* y *lookups* como se desee. Para crear el adaptador es necesario definir antes un archivo de configuración en el que se detallan los *comandos* y *lookups* que tendrá el adaptador. Una vez creado el adaptador posee ciertos métodos que conforman su ciclo de vida, los cuáles se detallan a continuación:

- `bool initialize()`: primer método que se ejecuta tras la llamada al constructor. En este método se inicializan variables y recursos que son necesarios en el adaptador.
- `bool start()`: arranca el adaptador para que comience su ejecución y pueda establecer enlaces de comunicación con los elementos del mundo exterior.
- `bool stop()`: detiene la ejecución del adaptador.
- `bool reset()`: resetea el adaptador en la fase previa a *start*.
- `bool shutdown()`: libera los recursos y finaliza el adaptador.

Además los adaptadores implementan otros dos métodos especiales con los que se realizan las comunicaciones con el mundo exterior, los *comandos* y los *lookup*.

- `void executeCommand(const LabelStr& name, const std::list<double>&args, ExpressionId dest, ExpressionId ack)`: envía un comando al sistema externo, para ello le pasa el nombre del comando, los parámetros y dos variables más que devuelven los resultados de

la ejecución al UE. Cuando en el plan de PLEXIL existe un nodo comando, el UE hace la llamada a este método indicando el nombre del comando y los parámetros asociados a él con el objetivo de realizar la comunicación con el exterior.

- `void subscribe(const State& state)`: sirve para suscribirse a un *lookup* asíncrono, en el que la interfaz avisa al UE cuando se registra algún cambio en la variable externa.

- `void unsubscribe(const State& state)`: elimina la suscripción del *lookup* asíncrono.

Para devolver los parámetros al sistema de ejecución, la librería tiene implementado el médodo `m_execInterface.notifyOfExternalEvent()` que funciona de forma transparente para el usuario.

Para que el UE pueda comunicarse con los adaptadores necesita archivos de configuración que le indiquen los *comandos* y *lookups* integrados en cada una de las interfaces. Estos archivos de configuración se explicarán en la sección 6.3.2.

6.3.2 Archivo de configuración de interfaces

Para que los planes de PLEXIL puedan comunicarse con el exterior se necesita que el UE pueda comunicarse con dichos sistemas externos a través de sus adaptadores. A priori el UE no tiene constancia de los *comandos* y *lookups* que tienen implementados sus adaptadores, por lo que necesita un archivo de configuración que le indique dicha información. Esto le permite al UE enviar la solicitudes de *comandos* y *lookups* a la función asociada a dicho *comando* o *lookup* implementada en el adaptador. Un ejemplo del archivo de configuración se muestra a continuación:

```
<Interfaces>
    <Adapter AdapterType="Adaptador_1">
    <CommandNames>
      comando_1,
      comando_2,
           .
           .
           .
      comando_n,
    </CommandNames>
    <LookupNames>
      lookup_1,
      lookup_2,
           .
           .
           .
      lookup_n,
    </LookupNames>
  </Adapter>
  <Adapter AdapterType="Adaptador_2">
       .
       .
       .
   </Adapter>
 </Interfaces>
```

6.4 Adaptadores de la arquitectura

Para este proyecto se han implementado dos adaptadores que permiten al UE comunicarse con las capas adyacentes al ejecutor. El adaptador TurtlecomAdapter realiza la comunicación con la capa funcional de la arquitectura MoBAr, mientras que el adaptador PDDLAdapter permite la comunicación con la capa superior de la arquitectura, el deliberador. A continuación se va a explicar cada uno de los adaptadores, mostrando los comandos implementados en cada uno de ellos, así como la parte del archivo de configuración asociada a cada adaptador.

6.4.1 Adaptador TurtlecomAdapter

El adaptador TurtlecomAdapter es el encargado de realizar la comunicación con la capa funcional (ver capítulo 5), enviando al robot las acciones que debe realizar. El adaptador traduce las órdenes del ejecutor a peticiones entendibles por los servidores RPC implementados en la capa funcional. Además de realizar la traducción, guarda información de estado necesaria para que el ejecutor mantenga la capa funcional dentro de los márgenes nominales de operación.

Para desarrollar este adaptador ha sido necesario incluir en el archivo de configuración de interfaces un nuevo apartado donde se definen los comandos implementados en este adaptador, y cuyo extracto se puede encontrar en las siguientes líneas:

```
<Adapter AdapterType="TurtlecomAdapter">
    <CommandNames>
        init_locomotion, moveto, rotate,
        checkPosition, checkAngle, bumpersensor,
        moveBack, moveBackxMeters, takePicture,
        movePtu, readBateryLevel, checkCharging,
        autodocking, stopAction, restartDockServer,
        restartPictureServer, finalPlan,
    </CommandNames>
  </Adapter>
```

A continuación se explican con mayor nivel de detalle cada uno de los comandos descritos den el archivo de configuración:

- `Command init_locomotion(String host, Integer sqsize, Integer offsetx, Integer offsety)`: comando que inicia todo sistema para establecer la comunicación con la capa funcional. Es necesario indicarle la dirección IP del host en el que está montada la capa funcional, el tamaño de las casillas del mapa por el que se moverá el robot, y el *offset* inicial que tendrá el robot dentro del mapa. Cuando se llama a este comando se inicializan todas las variables de estado necesarias para ubicar al robot en el mapa con respecto a los ejes de referencia, y se crean los clientes RPC que realizan la comunicación con la capa funcional.

- `String Command moveto(String pos)`: este comando mueve el robot a la posición del mapa indicada. La posición del mapa se le pasa al adaptador en un string con formato `cy_x`, dónde x e y son las coordenadas de la casilla a la que quiere ir el robot (este formato viene definido por el deliberador, veáse sección 7.2 para mas detalles). Cuando el robot finaliza el movimiento devuelve su posición en el mismo formato que la recibió.

- `String Command rotate(String ang)`: comando que sirve para girar el robot un ángulo determinado. Si el ángulo de giro es positivo el robot gira a la derecha, y si por el contrario es negativo, gira a la izquierda. El formato de dicho string es `a`*`angulo`* (este formato viene definido por el deliberador, veáse sección 7.2 para mas detalles). Como resultado devuelve la orientación del robot cuando finaliza al movimiento en el mismo formato que se envió el ángulo de giro.

- `String Command checkPosition()`: comando que devuelve la posición del robot en el mapa con formato `c`*`y_x`*.

- `String Command checkAngle()`: obtiene el ángulo de orientación del robot con respecto al sistema de referencia en formato `a`*`angulo`*.

- `Integer Command bumpersensor()`: comprueba el estado del sensor de parachoques y devuelve un entero indicando que parte del parachoques está pulsada. Si no hay ninguna zona en contacto con objetos devuelve cero.

- `String Command moveBack(Integer dismov)`: mueve hacia atrás el robot la distancia indicada en cm. Como resultado devuelve la posición del robot en el mapa al acabar el movimiento en formato `c`*`y_x`*.

- `String Command moveBackxMeters(String Orientation, String Robotsquare)`: comando que mueve hacia atrás el robot una determinada longitud en función de la orientación del robot. Devuelve la posición del robot en el mapa en formato `c`*`y_x`*, y se utiliza para mover hacia atrás el robot cuando encuentra algún obstáculo.

- `Boolean Command takePicture()`: este comando permite hacer una foto. Devuelve un booleano indicando si ha sido posible realizar la foto.

- `String Command movePtu(String yaw_pitch)`: comando que realiza un movimiento en la *pantilt* que tiene incorporada el robot. El modo de pasar los parámetros es similar al de los comandos anteriores, mediante un string con formato `p`*`_yaw_pitch`* que indica las posiciones a alcanzar por la *pantilt*. La función devuelve un string con la posición final de la *pantilt*, con la misma estructura que el string enviado.

- `Real Command readBateryLevel()`: permite leer el nivel de batería del robot, devolviendo su resultado en una variable de tipo Real.

- `Boolean Command checkCharging()`: comando que verifica si el robot está cargando o no. Si el resultado es falso significa que el robot no está cargando en su estación de carga y si el resultado es verdadero significa que el robot está cargándose.

- `Integer Command autodocking()`: inicia el proceso de búsqueda de la base de carga para que el robot inicie un proceso de recarga. Como resultado devuelve un entero que indica si el proceso se ha realizado con éxito o no. Un -1 significa que no se ha podido conectar con el servidor de recarga, un -2 que el proceso de búsqueda de la base falló y un 0 que la búsqueda se realizó correctamente.

- `Boolean Command stopAction(String action)`: manda una acción al modulo *safety* de la capa funcional para detender alguno de los bloques en ejecución. Envía un string con el módulo a detener (dock, locomotion o picture) y como resultado recibe un booleano con el resultado de la acción.

- `Boolean Command restartDockServer()`: comando para rearrancar el servidor de recarga que devuelve un booleando con el resultado de la operación.

- `Boolean Command restartPictureServer()`: comando que reinicia los servicios de la cámara en la capa funcional cuyo resultado es un booleano que indica si la operación ha sido satisfatoria.
- `Boolean Command finalPlan()`: comando que indica la finalización del plan.

6.4.2 Adaptador PDDLAdapter

El segundo adaptador implementado comunica ejecutor y deliberador (ver capítulo 7) y permite instanciar tantos planificadores como se desee, ya que cada planificador tendrá un identificador único. Al igual que en el adaptador anterior, ha sido necesario añadir en el archivo de configuración de interfaces los comandos implementados en este adaptador, mostrándose la configuración del adaptador en el siguiente fragmento de código:

```
<Adapter AdapterType="PDDLAdapter">
    <CommandNames>
      connect_planner_to_ogate, create_planner, get_plan,
      next_action, insert_predicate, modify_predicate,
      remove_predicate, predicate_object, modify_function,
      function_value, action_object, action_duration,
      put_obstacle, delete_goals,
    </CommandNames>
  </Adapter>
```

Siguiendo la misma estructura que en el adaptador anterior, a continuación se van a explicar con mayor nivel de detalle cada uno de comandos del interface PDDLAdapter:

- `Boolean Command connect_planner_to_ogate(String name, String host, Integer port)`: conecta el ejecutor con una herramienta *software* que ofrece soporte para testear sistemas de control autónomo a través de una interfaz gráfica, explicada en el Apéndice C. Para realizar la conexión es necesario enviar el nombre del ejecutor, la dirección IP del host en el que está ubicado OGATE y el puerto de conexión. Si la conexión ha sido satisfactoria devuelve un valor verdadero.
- `Integer Command create_planner(String configfile)`: crea un planificador y le asigna un identificador único. Devuelve el identificador del planificador.
- `Boolean Command get_plan(Integer numplanner, Boolean replan)`: comando que genera un plan para ser ejecutado por el ejecutor (véase sección 7.4). Además permite generar nuevos planes gracias a la incorporación de la variable replanificación. El resultado del comando es verdadero si el plan se generó correctamente, o falso si el plan generado no es válido.
- `String Command next_action(Integer numplanner)`: lee la siguiente acción del plan para ser ejecutada.
- `String Command action_object(Integer numplanner, Integer numobject)`: comando que obtiene el objeto *numobject* de la última acción leída.
- `Real Command action_duration(Integer numplanner)`: comando que obtiene la duración de la última acción leída.

- `Boolean Command insert_predicate(Integer numplanner, String predicate, String objects, Boolean isgoal)`: inserta un predicado en el problema PDDL pasandole el predicado y los objetos asociados a dicho predicado (véase seccion 7.2 para más detalles sobre PDDL). Al finalizar indica si la instrucción se ha insertado de forma satisfactoria.

- `Boolean Command modify_predicate(Integer numplanner, String predicate, String objectsold, String objectsnew)`: comando similar al anterior que modifica uno de los predicados del problema.

- `Boolean Command remove_predicate(Integer numplanner, String predicate, String objects)`: este comando elimina del problema PDDL el predicado indicado. Si la acción ha sido correcta retorna un valor verdadero.

- `String Command predicate_object(Integer numplanner, String predicate, Integer numobject)`: comando que solicita objeto número *numobjet* de la lista de objetos del predicado *predicate*.

- `Boolean Command modify_function(Integer numplanner, String predicate, String objects, Real value)`: modifica los parámetros objetos de una función del problema PDDL.

- `Real Command function_value(Integer numplanner, String predicate, String objects)`: devuelve el valor que tiene la función en el problema PDDL.

- `Boolean Command put_obstacle (String position, String angle, Integer downbumper)`: comando que añade un obstáculo en el mapa.

- `Boolean Command delete_goals()`: elimina todas las metas del problema PDDL.

6.5 Planes de PLEXIL para la arquitectura

En este apartado se va a describir el diseño del plan implementado en lenguaje PLEXIL que controlará todas las operaciones del robot. Este plan tiene comunicación con la capa deliberativa y con la capa funcional a través de los comandos implementados en los adaptadores de la sección anterior. El objetivo es que el ejecutor solicite a la capa deliberativa un plan a ejecutar por el robot para ser interpretado por el ejecutor, desgranando cada una de las acciones de alto de nivel en órdenes entendibles por la capa funcional. El ejecutor leerá secuencialmente todas las acciones del plan realizando las correspondientes traducciones hasta que finalmente no le queden más acciones por leer, momento en el que dará por finalizada la ejecución del plan. Además, si durante la ejecución del plan surge algún imprevisto, el ejecutor tratará de resolverlo para que el robot alcance los objetivos fijados inicialmente. Para entender mejor el modelo se ha creado un diagrama de flujo básico (figura 6.1) en el que se describen los pasos que sigue el ejecutor durante su ciclo de vida.

Como se puede ver en el diagrama de flujo, el ejecutor en primer lugar inicializa todas las variables instaciadas y conecta con la herramienta OGATE con el objetivo de que el operador pueda monitorizar la ejecución del plan. Después inicializa los adaptadores que realizan la comunicación con las dos capas adyacentes, para finalmente crear el planificador y obtener el primer plan. Si la generación del plan no es correcta lo vuelve a intentar dos veces más y si estos dos intentos vuelven a fallar el ejecutor le ordena al robot que se dirija a la estación de carga finalizando así la ejecución del plan. Si por el contrario la creación del plan es válida se leen de forma secuencial todas las acciones planificadas actuando en consecuencia

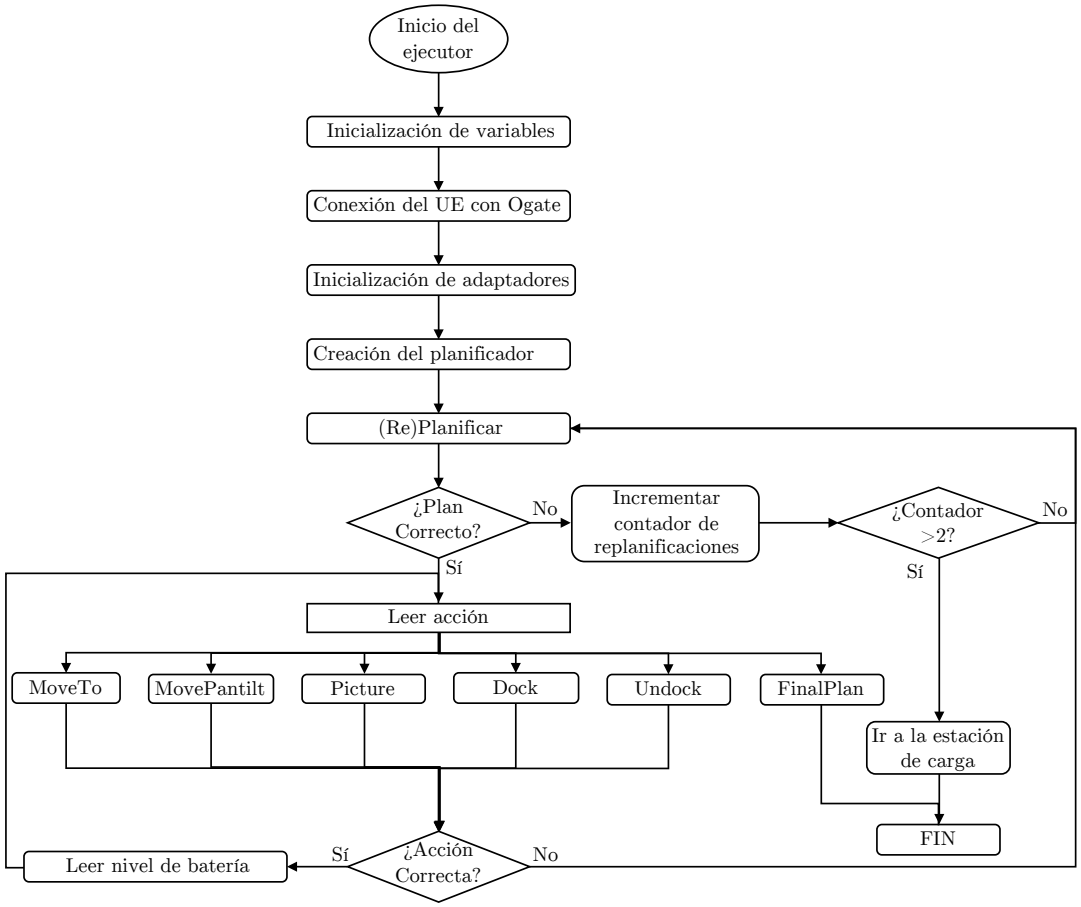


Figura 6.1: Diagrama de flujo plan general de PLEXIL

según sea la orden. Una vez que se ha ejecutado la orden, se verifica si su realización ha sido correcta. Si el resultado es correcto, se actualiza el nivel de batería y se lee la siguienta acción. Si por el contrario el resultado de la acción no es correcto, se actualizan los datos y se replanifica para obtener un nuevo plan y cumplir con las metas fijadas inicialmente. El proceso de lectura de nuevas acciones cotinúa hasta que no queden más acciones por leer, momento en el que finalizará la ejecución del plan. Las acciones que lee el planificador no han sido desarrolladas en el diagrama de flujo debido a la extensión en la representación del diagrama, por lo que su descripción se hará de forma escrita:

1. **MoveTo:** acción de alto nivel que indica la posición del mapa a la que desea mover el robot. La secuencia de operaciones es la siguiente
   (a) Descomponer la acción de alto nivel leída del plan para obtener la casilla destino a la que se desea mover el robot.
   (b) Mover el robot a la casilla deseada.
   (c) Actualizar el problema PDDL con la posición devuelta por el robot.
   (d) Comprobar que la posición devuelta por el robot se corresponde con la casilla destino.
      i. Si es correcto se marca la acción como correcta y se pasa a leer la siguiente acción.
      ii. Si no es correcto es que el robot ha detectado un obstáculo.
         A. Añade el obstáculo en el mapa.
         B. Mueve hacia atrás el robot para dejarlo en una zona segura.
         C. Actualiza la nueva posición en el problema PDDL.

D. Marca la acción como fallida para que el sistema replanifique

Además, en paralelo hay un temporizador que controla el tiempo que dura la acción de movimiento. Si se supera el tiempo del temporizador se eliminan todas las metas del plan y se envía al robot a la posición de salida. Este tiempo viene dado por por el plan generado en el deliberador.

2. **MovePantilt:** acción que mueve la *pantilt* en las posiciones de *pitch* y *yaw* indicadas. El proceso es el siguiente:

   (a) Se descompone la acción de alto nivel para extraer la posición de pitch y yaw.

   (b) Mueve la *pantilt* a la posición deseada.

   (c) Actualiza el problema PDDL con el resultado obtenido.

   (d) Comprueba la posición final de la *pantilt*.

      i. Si es correcta se marca la acción como correcta y se pasa a leer la siguiente acción.

      ii. Si no es correcta se repite el movimiento.

3. **Picture:** acción de alto nivel que toma una fotografía orientada y que se corresponde con una meta del problema PDDL. Las tareas que se llevan a cabo para descomponer la acción son las siguientes:

   (a) Descompone la acción de alto nivel obteniendo la casilla en la que se toma la foto y la orientación de la *pantilt*.

   (b) Se realiza la captura de la fotografía.

   (c) Se inserta un nuevo predicado en el el problema PDDL para indicar que se ha cumplido el goal.

   En paralelo existe un temporizador que controla el tiempo que tarda la capa funcional en capturar imágenes, por lo que si salta este temporizador significa que los servicios de bajo nivel de los que hace uso la cámara están caídos. Si esto ocurre el ejecutor ordena a la capa funcionar rearrancar dichos servicios y vuelve a intentar la captura de imagen. El tiempo viene fijado en el plan generado por el deliberador.

4. **Dock:** acción que inicia la búsqueda de la estación de carga. Las tareas en las que se decompone esta acción se detallan a continuación:

   (a) Se establece comunicación con el servidor de recarga.

      i. Si falla la comunicación se rearranca el servidor.

   (b) Se inicia la búsqueda de la estación de carga.

      i. Si tarda más de 10 segundos el robot no ha orientado bien, por lo que:

         A. Se detiene la búsqueda.

         B. Se echa hacia detrás el robot para que tenga más probabilidad de reorientarse hacia la base.

         C. Se vuelve a iniciar la búsqueda repitiendo el proceso.

   (c) Se actualiza el estado del robot a *cargando* en el problema PDDL.

   También se encuentra trabajando en paralelo un temporizador que salta si la acción tarda demasiado. Cuando salta el temporizador se detiene la acción de búsqueda de la estación de carga, se actualiza la posición del robot en el problema PDDL y se marca la acción como errónea para forzar la replanificación. Este tiempo viene dado por por el plan generado en el deliberador.

5. **Undock:** acción que aleja al robot de la estación de carga. Los pasos que se realizan para alejarlo son los siguientes:

   (a) Se le ordena al robot retroceder el tamaño de una casilla del mapa.

   (b) Se actualiza la posición del robot en el problema PDDL.

   Además se incluye un temporizador que trabaja en paralelo y que detiene la acción si ésta tarda demasiado, al detener la acción se actualiza la posición del robot en el problema PDDL y marca la acción como fallida para forzar la replanificación.

6. **FinalPlan:** acción de la última línea del plan que indica la finalización del mismo. Cuando se lee esta acción se envía un comando a los adaptadores para que liberen los recursos.

Capítulo 7

El deliberador

En este capítulo se presenta el nivel superior de la arquitectura, el deliberador. Se comienza con una introducción de las técnicas de planificación y scheduling. La siguiente sección describe el lenguaje de modelado de planes de alto nivel, Plan Domain Definition Languaje (PDDL). En la tercera parte del capítulo se muestra el modelado del entorno realizado para este proyecto, dejando para el último apartado la librería que interactúa con el planificador.

7.1 Introducción

El deliberador es la capa de mayor nivel de abstracción de la arquitectura de control autónoma MoBAr. Su objetivo es encontrar una secuencia óptima de acciones que conduzcan al robot a un estado deseado partiendo de un estado inicial. Para obtener esta secuencia óptima de acciones se hace uso de los procesos de planificación y scheduling. La planificación se encarga de obtener las acciones necesarias que consigan cumplir las metas establecidas, mientras que el scheduling se encarga de los recursos que son necesarios para llevar a cabo dichas acciones. No obstante, la selección óptima de acciones está condicionada por restricciones numéricas propias del scheduling, lo que hace que estas dos disciplinas se complementen perfectamente y que en la actualidad se encuentren integradas. Además, dichas técnicas son fácilmente aplicables a cualquier problema de optimización de recursos que se pueda encontrar en la vida real, cuyo ejemplo más representativo se corresponde con una empresa de reparto en la que se ha de distribuir material en una determinada zona geográfica con unos recursos limitados (tiempo de entrega, número de empleados, número de camiones, costes, etc.). Con la aplicación de las técnicas de planificación y scheduling se espera obtener la ruta óptima que minimice el tiempo de entrega, maximizando el beneficio neto de la empresa. En este capítulo ambas técnicas se van utilizar en el ámbito de la robótica, dónde se realizará un modelado del comportamiento de un robot así como un modelo del mundo que lo rodea. En dicho modelo se establecerán las acciones básicas del robot, sus restricciones y estados iniciales, así como las metas que se desean alcanzar, para que un planificador genere un plan óptimo teniendo en cuenta las restricciones, objetivos y estados del modelo.

Para realizar el modelado del problema de planificación se va a hacer uso del lenguaje Plan Domain Definition Language (PDDL), descrito con detalle en la siguiente sección.

7.2 Plan Domain Definition Language

El lenguaje PDDL es el estandart en lenguajes de planificación. Su desarrollo vino de la mano de Drew McDermott en el año 1998, empleándose por primera vez en la Competición de Planificación Internacional (IPC) del mismo año [36]. Dicho lenguaje separa el modelo de planificación en dos partes: una descripción del dominio y una descripción del problema, permitiendo una separación entre elementos: el dominio incluye los elementos comunes del modelo (describe el comportamiento del robot y del mundo que lo rodea), mientras que el problema contiene los elementos particulares de cada modelo (características típicas de cada robot y mundo). Sin embargo la versión inicial de PDDL no ha sido la definitiva, ya que han aparecido modificaciones que incorporen nuevas características:

- PDDL 2.1[37]: introduce funciones numéricas (modelo distinto al binario que permite medir niveles, distancias, tiempos, etc.) y la creación de métricas temporales.
- PDDL 2.2 [38]: dota de mayor capacidad el modelado de problemas reales intorduciendo predicados derivados y literales que marcan los tiempos de las acciones.
- PDDL 3.0 [39]: incluye restricciones que permiten omitir objetivos.

Una vez hecha la introdución histórica del lenguaje PDDL, se va a profundizar en cada una de las dos partes de las que se compone un problema de planificación PDDL: el dominio y el problema:

- **Dominio:** contiene los requisitos del entorno así como las acciones que actúan sobre dicho entorno y en el que además se pueden definir funciones. Se compone de dos elementos básicos:
  - Predicados: el predicado es el elemento principal de PDDL sobre el que trabajan las acciones. Las acciones modifican los predicados para buscar la solución al problema. Además cada predicado puede estar compuesto por uno o más objetos. Análogamente, se puede comparar el predicado con un estado del problema a resolver.
  - Acciones: es el componente que permite modificar el estado (modifica predicados). La complejidad de las acciones viene determinada por la versión de PDDL y por la cantidad de precondiciones y efectos que contenga. Las precondiciones son una serie de condiciones (expresadas con predicados) que se deben cumplir para que una acción se lleve a cabo, mientras que lo efectos son los predicados que hay que añadir o eliminar una vez que se ha llevado a cabo la acción.
- **Problema**: contiene las características propias de cada modelo. En él se encuentra el estado inicial del mundo y las metas a cumplir. Para ello define todos los objetos que modelarán el problema y que vienen definidos por los predicados descritos en el dominio PDDL. También tiene que incluir las metas que se desean cumplir formadas por predicados definidos previamente en el dominio. Para generar un plan válido el planificador debe comprobar que el estado final del mundo contiene los predicados metas.

A partir de estos dos archivos con extensión `.pddl` un planificador PDDL obtendrá una serie de acciones en base a los objetivos especificados y al estado inicial del mundo. Para ello el planificador irá generando una serie de acciones válidas (es decir, que cumpla las precondiciones) que den como resultados los objetivos marcados.

7.3 Modelado del entorno

El modelado del problema se realiza a través de la definición de predicados y acciones en el archivo de dominio y la instanciación de objetos y fijado de metas en el problema PDDL. Esta sección tiene como objetivo dar a conocer cada uno de los dos ficheros explicando todos los elementos que definen el comportamiento del robot. Mediante este modelado se pretende conseguir que el robot realice una serie de acciones en determinados puntos de un mapa, obteniendo una ruta que pase por dichos puntos, aunque para ello es necesario tener una serie de consideraciones previas:

- El entorno queda modelado por un mapa en dos dimensiones dividido en celdas al igual que un tablero de ajedrez. En dicho mapa existen dos tipos de casillas: libres y con obstáculos, siéndo las casillas libres las únicas por las que puede circular el robot. Cada celda viene definida por sus coordenadas (x,y) con respecto a sus ejes de referancia.
- En el mapa puede haber casillas especiales en las que se realice una función específica como por ejemplo puntos de recarga.
- El robot queda definido por su posición en el mapa, la orientación de la *pantilt*, una variable que indica si se encuentra cargando o en estado de descarga y el nivel de batería.
- El robot puede realizar una serie de acciones: movimiento entre casillas, movimiento de la *pantilt*, captura de fotografías, inicio de carga automática, recarga de batería y finalización de la carga.
- Todas las acciones tienen un coste temporal y energético.

Con todas estas consideraciones ya se pueden definir los archivos necesarios para modelar un problema en lenguaje PDDL.

7.3.1 Dominio

El dominio, como ya se ha explicado anteriormente, contiene la definición de predicados y funciones así como la descripción de las acciones. Para nuestra aplicación se han desarrollado los siguientes predicados:

- *Position*: predicado que contiene la posición del robot en el mapa.
- *Dock*: predicado que contiene la posición de la estación de carga en el mapa.
- *Ptu_pos*: predicado que indica la orientación de la *pantilt*.
- *Picture*: predicado empleado para capturar fotografías en una posición del mapa y con una determinada orientación de la cámara.
- *Is_charging*: predicado que indica si el robot se encuentra en estado de recarga.
- *Free*: predicado que indica si el robot es libre para moverse.

Las acciones creadas para definir el comportamiento del robot se detallan a continuación:

- *Charge*: acción que carga la batería del robot. Las precondiciones comprueban que el robot se encuentra en la casilla que contiene la estación de carga y que además está conectado a dicha base. El efecto de esta acción consiste en incrementar la energía del robot.

- *Autodock*: acción que conecta el robot a la base de carga para iniciar un ciclo de carga. Las precondiciones comprueban que el robot tiene capacidad de movimientos (predicado *free*) y se encuentra ubicado en la celda que contiene la estación de carga. Su efecto consiste en bloquear al robot (eliminar predicado *free*) poniéndolo en estado de carga (añadiendo el predicado *is_charging*). Además decrementa la energía del robot debido a que esta acción requiere un consumo de energía.

- *Undock*: acción que desconecta el robot de la base de carga. Las precondiciones que se deben cumplir son que el robot se encuentre en la casilla de la estación de carga y que además esté cargando su batería (predicado *is_charging*). Los efectos de esta acción consisten en dejar libre al robot (eliminando el predicado *is_charging* y añadiendo el predicado *free*) y decrementar su nivel de batería.

- *MovePtu*: acción que orienta la *pantilt* en la posición de *pitch* y *yaw* especificadas en el predicado *ptu_pos*. Antes de iniciar la acción se comprueba que el robot es libre (predicado *free*) y que la energía del robot está por encima del nivel requerido para realizar la acción. El efecto que presenta esta acción consiste en la actualización del predicado *ptu_pos* con la nueva orientación y el decremento del nivel de batería.

- *TakePicture*: acción que toma una fotografía orientada en la casilla del mapa indicada. Las precondiciones comprueban que el robot se encuentra libre en la casilla adecuada con la *pantilt* orientada correctamente. Tras realizar esta acción añade un nuevo predicado *picture* en el estado actual. Además efectúa un decremento de la batería.

- *MoveTo*: acción que mueve el robot entre dos casillas. Antes de iniciar la acción se comprueba que el robot está libre y que la *pantilt* se encuentra orientada mirando al frente. Cuando termina la acción se modifica el predicado *position* con la nueva posición y se decrementa el nivel de batería del robot en función de la distancia recorrida.

Además el dominio incluye funciones para el cálculo de la distancia entre casillas, y el consumo de energía de los distintos elementos (*pantilt*, avance, cámara, estación de recarga, ...) así como la duración que tiene cada uno de los componentes.

7.3.2 Problema

El problema contendrá los elementos que definen cada modelo particular. El archivo generado se estructura en tres bloques claramente definidos, los cuales se detallan a continuación.

- *Definición de objetos*: primera parte del archivo en el que se definen los diferentes objetos del modelo (posiciones posibles de la *pantilt*, casillas del mapa, robot, etc.).

- *Inicialización de predicados y funciones*: segunda parte del archivo en el que se instancian las funciones y predicados definidos en el dominio. Aquí se encuentra reflejada la distancia entre las distintas casillas, el nivel inicial de batería, el tamaño de las casillas, la posición y el estado inicial del robot, la energía inicial y los consumos de energía.

- *Establecimiento de metas*: en la parte final del arhivo se definen las metas que debe cumplir el robot. Aquí se incluyen distintos puntos por los que se desea que pase, posiciones en las que capturan imágenes, recargas, ...

Con todos los componentes incluidos en ambos archivos (dominio y problema) queda definido el comportamiento final del robot, el cual podrá moverse por un mapa para realizar fotografías orientadas o dirigirse a la estación de carga. Además, gracias a la sepación de archivos, se pueden generar varios escenarios de forma rápida ya que basta con modificar el problema para tener escenarios completamente distintos (cambiar la posición inicial del robot, cambiar consumos, cambiar las metas, etc.).

7.4 Librería de Planificación

La generación de planes se lleva a cabo por un programa planificador que soporte la versión de 3 de PDDL. Este planificador leerá el dominio y el problema PDDL desarrollados en la sección anterior y generará un plan que cumpla con las metas especificadas el el problema PDDL, teniendo en cuenta el estado inicial del mundo y las acciones posibles que puede llevar a cabo el robot. Para poder llevar a cabo la ejecución, lectura y modificación del entorno de modelado PDDL se ha proporcionado una librería en C++ ya desarrollada (figura 7.1) que permite manejar un planificador PDDL de forma transparente, porporcionando los servicios necesarios al adaptador *PDDLAdapter* definido en la sección 6.4.2

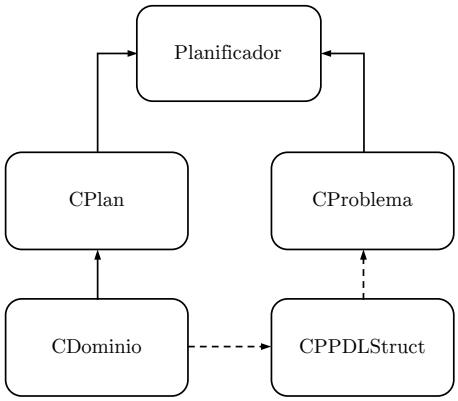


Figura 7.1: Diagrama de clases de la librería que controla el planificador

- CPDDLStruct: contiene la definición de estructuras necesarias para almacenar la información de predicados, acciones y funciones de los archivos PDDL.
- CDominio: tiene acceso a un vector con la información de las acciones que posee el dominio.
- CProblema: permite la insercción o modificación de las metas, estados iniciales o funciones del problema PDDL.
- Cplan: lee el plan generado por el planificador a través de un archivo que almacena dicho plan.
- Planificador: instancia un planificador genérico y comprueba que se ejecuta correctamente mediante los datos obtenidos en un fichero de configuración.

Capítulo 8

Resultados experimentales

En este capítulo se van a mostrar los resultados obtenidos tras realizar 6 escenarios de pruebas en los que se evalúan distintos comportamientos que debe tener el Turtlebot II bajo el control de la arquitectura MoBAr. La primera sección describe el escenario de pruebas por el que se moverá el robot y explica brevemente los planificadores empleados, mientras que las secciones sucesivas se encargan de explicar cada una de las pruebas realizadas.

8.1 Escenario de pruebas

Todas las pruebas se van a realizar en el laboratorio E-31 de la Escuela Politécnica Superior. Su distribución es perfecta para realizar las pruebas, ya que estamos ante un escenario acotado con suficiente espacio para mover el robot y que además cuenta con obstáculos perfectamente distribuidos (mesas de trabajo, sillas, papeleras y armarios). Para poder realizar las pruebas se ha elaborado un mapa con la distribución del laboratorio, en el que se reprensentan casillas libres por las que puede moverse el robot y casillas ocupadas que tienen obstáculos fijos (mesas y armarios). Las dimensiones del mapa son de 7,6 x 6,8 m, dividido en celdas rectangulares de 0,4 x 0,4 m, lo que da un total de 19 x 17 celdas. El tamaño de las casillas se fijó teniendo en cuenta el tamaño de las baldosas del suelo para que coincidiesen con las celdas del mapa, pudiéndose elaborar rápidamente un mapa de la superficie del laboratorio. Además este tamaño de celda es ligeramente superior al diámetro de la base Kobuki por lo que el robot encaja perfectamente centrado en una baldosa, pudiendo rotar sin salir de los límites de la misma.

Una representación gráfica del mapa se puede encontrar en la figura 8.1, dónde las casillas sombreadas representan casillas que tienen obstáculos permanentes, mientras que las casillas sin colorear son las zonas por las que el robot puede moverse libremente. Además, en dicha figura aparece una casilla con un color más oscuro, indicando que dicha celda contiene la estación de recarga.

Se van a utilizar dos planificadores basados en PDDL y asociados con un algoritmo de planificación de rutas (*path-planning*) que se encarga de definir los movimientos del robot por el mapa. Los planificadores usados varían en función de la integración de dicho algortimo con el planificador PDDL como se detalla a continuación:

- **OPTIC$_{S\theta}$:** planificador que integra por separado el algoritmo de planificación rutas y el algoritmo de planificación de tareas PDDL , ejecutando en primer lugar el algoritmo de planificación de rutas, el cuál obtiene la distancia a los objetivos teniendo en cuenta la posición inicial del robot y la posición de cada una de las metas. Estas distancias son incluídas en el problema PDDL en forma

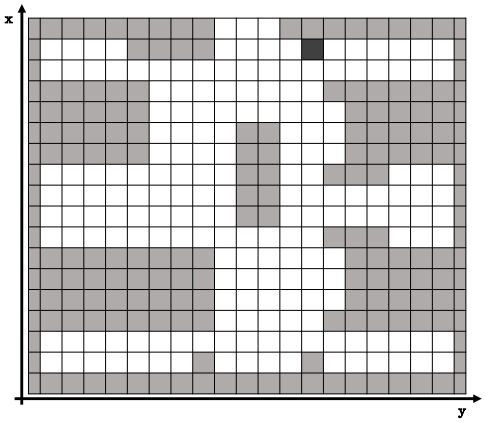


Figura 8.1: Mapa realizado del laboratorio E-31

de función para que el planificador PDDL obtenga el conjunto final de acciones a realizar por el robot.

- **UP2TA**[40]: integra de manera conjunta las funciones planificación de rutas y tareas, incluyendo en la heurística de búsqueda del planificador la distancia entre objetivos. De esta forma el planificador evalúa en su conjunto las acciones a planificar y las posibles rutas obteniendo presumiblemente mejores resultados.

Para el cálculo de rutas en ambos planificadores se ha empleado el algortimo Theta* Smooth (S-Theta*) [41], un algoritmo de búsqueda de rutas que al no tener restricciones de giro obtiene mejores rutas que otros algoritmos como A* [42]. Sin embargo ambos planificadores no hacen uso del mismo planificador PDDL, ya que OPTIC$_{S\theta}$ utiliza el planificador OPTIC [43], mientras que UP2TA emplea el planificador FF [44].

8.2 Primer banco de pruebas

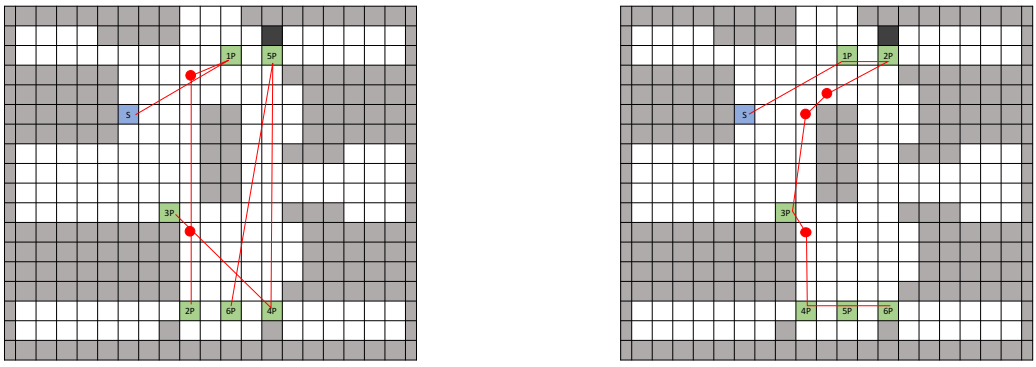
Este primer banco de pruebas consiste en la adquisición de 6 imágenes en distintos puntos del laboratorio E-31. La prueba medirá el tiempo de búsqueda de la ruta, el tiempo de ejecución del plan, la distancia recorrida calculada por el planificador y la distancia recorrida calculada con los sensores del robot. La prueba se realizará con los dos planificadores expuestos en la sección anterior, realizando 3 repeticiones en cada uno de ellos, con el objetivo de tener una media de resultados.

Los resultados recogidos en la tabla 8.1 muestran como OPTIC$_{S\theta}$ es más rápido obteniendo el plan, pero UP2TA consige ejecutar el mismo plan en menor tiempo debido a la optimización de ruta conseguida con la integración de los algoritmos de planificación de rutas y tareas. En cuanto a la distancia planificada y la distancia medida con los sensores del robot, ésta varía entre un 1 y un 2 %, debido a acciones correctoras de la capa funcional que no se tienen en cuenta en el modelo PDDL de la capa deliberativa, lo que significa que el modelo PDDL se acerca bastante al robot real.

Tabla 8.1: Resultados recogidos tras la ejecución del primer banco de pruebas

Planificador	**Tiempo de búsqueda (ms)**	**Tiempo de ejecución (s)**	**Distancia planificada (m)**	**Distancia medida (m)**
OPTIC$_{S\theta}$	68	370	23.57	23.32
UP2TA	308	251	11.10	11.41

La trayectorias obtenidas por los planificadores quedan representadas en la figura 8.2, en ella se puede ver como ambos escenarios parten de la casilla de salida (marcada en azul con una S), y tienen 6 puntos en los que se realizan fotografías (marcadas en verde y numeradas del P1 al P6). En ambas trayectorias aparecen casillas con puntos rojos que indican que el robot se ha detenido en esa casilla para realizar un cambio de trayectoria. Sin embargo, la trayectoria recorrida por los dos planificadores no es la misma, ya que la trayectoria seguida por OPTIC$_{S\theta}$ (figura 8.2a) realiza un mayor recorrido provocado por la desorganización a la hora de ordenar las tareas por parte del planificador, traducido en un mayor consumo de batería y tiempo de ejecución. En cambio, la trayectoria que obtiene UP2TA (figura 8.2b) consigue recorrer los objetivos de una forma ordenada, reduciendo el tiempo de ejecución y mejorando la autonomía del robot. De hecho UP2TA consigue reducir la distancia recorrida por el robot en un 53 %, disminuyendo así el tiempo de ejecución en un 32 % con respecto a su competidor OPTIC$_{S\theta}$, mejoras conseguidas gracias a la integración conjunta de los algoritmos de planificación de tareas y rutas.



(a) Trayectoria seguida con OPTIC$_{S\theta}$ (b) Trayectoria seguida con UP2TA

Figura 8.2: Trayectoria recorrida por el robot en el primer banco de pruebas

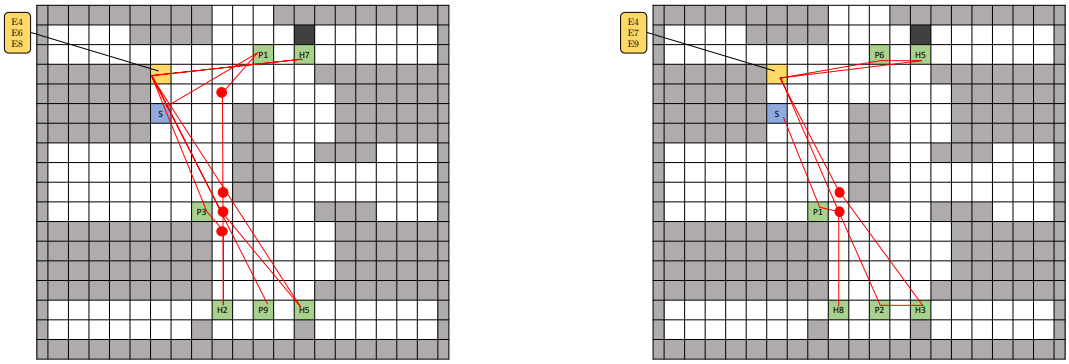
8.3 Segundo banco de pruebas

Para contrastar los buenos resultados obtenidos por UP2TA se ha construido este nuevo banco de pruebas aumentando la complejidad del problema a resolver. El escenario sobre el que se desarrolla la prueba es el mismo, pero ahora el robot realizará dos tipos distintos de fotografías: fotografías de baja calidad (iguales que las del ejemplo anterior), y fotografías de alta calidad. El robot puede contener en su interior todas las fotografías de baja calidad que desee, pudiendo poseer únicamente una fotografía de alta calidad. Si desea realizar más fotografías de alta calidad antes debe de ir a descargar la anterior a un punto habilitado para ello. En este segundo banco de pruebas el robot partirá desde el mismo punto de partida que en el ejemplo anterior y realizará 3 fotografías de baja calidad y 3 fotografías de alta calidad, por lo que tendrá que realizar descargas intermedias para poder realizar las fotografías de alta calidad. Los puntos de captura fotográfica son los mismos que en el ejemplo anterior, situando el punto de descarga dos celdas por encima de la posición de partida.

Para la obtención de resultados se utilizan los mismos planificadores y las mismas métricas que en la prueba anterior, quedándo recogidas las trayectorias de los planificadores en la figura 8.3 y los resultados de la prueba en la tabla 8.2. Para seguir la misma representación simbólica que en la prueba anterior la casilla de salida está representada en azul, mientras que las casillas de captura fotográfica quedan en color verde, distinguiendo entre fotografías de baja calidad (P) y fotografías de alta calidad (H) junto con la incorporación de la nueva casilla de descarga en color amarillo (E).

Tabla 8.2: Resultados recogidos tras la ejecución del segundo banco de pruebas

Planificador	Tiempo de búsqueda (ms)	Tiempo de ejecución (s)	Distancia planificada (m)	Distancia medida (m)
OPTIC$_{S\theta}$	138	582	34.93	35.29
UP2TA	5041	480	26.55	26.82



(a) Trayectoria seguida con OPTIC$_{S\theta}$

(b) Trayectoria seguida con UP2TA

Figura 8.3: Trayectoria recorrida por el robot en el segundo banco de pruebas

Al igual que ocurría en el experimento anterior, las trayectorias obtenidas por los planificadores distan bastante entre sí. La trayectoria calculada por OPTIC$_{S\theta}$ (8.3a) primero captura la imagen de bajo nivel situada arriba a la derecha (P1), luego se dirige a por la fotografía de alta calidad situada en la parte inferior derecha del mapa (H2), coje otra fotografía de baja calidad (P3) y se dispone a descargar la fotografía de alta calidad en el punto de descarga (E4). Una vez descargada la fotografía se dirige a por la fotografía de alta calidad situada en la parte inferior derecha del mapa (H5), la descarga (E6) y procede a capturar la última imagen de alta calidad (H7). Descarga dicha fotografía (E8) y se va a por la última imagen de bajo nivel (P9). Sin embargo UP2TA (figura 8.3b) realiza un recorrido menos caótico ya que en primer lugar recoge la fotografía más próxima por la parte inferior de la casilla de salida (P1), para posteriormente recoger P2 y H3. Como tiene una fotografía de alta calidad la lleva al punto de descarga (E4) y pasa a recoger H5. Antes de descargar esta última fotografía de alta calidad captura la imagen que tiene de camino (P6). Por último descarga la fotografía que posee de alta calidad (E7) y se dirige a por la última fotografía de alta calidad (H8) para dejarla posteriormente en el punto de descarga (E9). Este recorrido menos caótico se traduce en que UP2TA reduce un 24 %, la distancia recorrida con respecto a OPTIC$_{S\theta}$, disminuyendo también el tiempo de ejecución en un 17,5 %. Al igual que en el experimento anterior esta reducción de distancia y tiempo de ejecución se traduce en un ahorro energético que proporciona mayor autonomía al robot. Sin embargo, el tiempo de búsqueda de la solución al problema en este caso es muy superior en el planificador UP2TA (36 veces superior aproximadamente), aunque este planificador sale como ganador gracias a que la suma del tiempo de búsqueda de solución sumado al tiempo de ejecución es menor (582.138 s para OPTIC$_{S\theta}$ frente a 485.04 s para UP2TA).

8.4 Tercer banco de pruebas

Tras la realización de los dos primeros bancos de pruebas se incorporaron mejoras en la capa funcional. Estas mejoras modifican la gestión de topics (cambiando el método de suscripción para recibir los datos en

menos tiempo) y se incorporó el algoritmo PI descrito en la sección 5.4.2. Anteriormente el robot realizaba una suscripción a los topics esperando durante un tiempo fijo, mientras que ahora en cuanto está el dato disponible es recibido por el nodo que lo solicita. En cuanto al algortimo PI, con su integración se espera mejorar la velocidad y la precisión de los movimientos, gestionados anteriormente por un mecanismo de control precario. Para comprobar que estas incorporaciones mejoran las capacidades de la arquitectura de control se van a repetir los experimentos realizados en el primer y segundo banco de pruebas, recogiendo las mismas métricas. En este apartado no se va a analizar la trayectoria seguida por el robot puesto que es la misma que en los bancos de pruebas anteriores (misma ejecución del plan sin cambiar ningún parámetro).

Al igual que en los dos apartados anteriores, se realizan 3 repeticiones de cada prueba, con el objetivo de tener valores medios. Los resultados obtenidos tras la repetición de primer banco de pruebas quedan recogidos en la tabla 8.3.

Tabla 8.3: Resultados recogidos tras la repetición del primer banco de pruebas

Planificador	**Tiempo de búsqueda (ms)**	**Tiempo de ejecución (s)**	**Distancia planificada (m)**	**Distancia medida (m)**
OPTIC$_{S\theta}$	69.57	263	23.31	23.53
UP2TA	337.40	205.66	11.10	11.35

Si se comparan los resultados de la tabla 8.3 con los de la tabla 8.1, se observa que los los tiempos de ejecución para cada planificador mejoran notablemente (en torno a un 23 % cada uno), mientras que la distancia recorrida disminuye ligeramente (menos de un 1 %), resultados esperados tras las mejoras de la capa funcional. Sin embargo, los tiempos en la búsqueda del plan empeoran entre un 1 y un 9 % debido a la ejecución de procesos en segundo plano que penalizaron al planificador en la búsqueda de la solución.

Para confirmar los buenos resultados recogidos tras la repetición del primer banco de pruebas, se va a hacer lo mismo con el segundo, volviendo a ejecutar el segundo banco de pruebas incluyendo las mejoras en la capa funcional. Tras realizar las 3 repeticiones pertienentes se obtuvieron los resultados reflejados en la tabla 8.4.

Tabla 8.4: Resultados recogidos tras la repetición del segundo banco de pruebas

Planificador	**Tiempo de búsqueda (ms)**	**Tiempo de ejecución (s)**	**Distancia planificada (m)**	**Distancia medida (m)**
OPTIC$_{S\theta}$	169.41	454	34.93	35.19
UP2TA	6539.1	372	26.55	26.82

Como era de esperar, los resultados de la tabla 8.4 mejoran los resultados del segundo banco de pruebas (tabla 8.2). El tiempo de ejecución mejora en torno a un 22 %, mientras que la distancia recorrida se mantiene invariable (variaciones menores de un 0.3 %). Al igual que en la prueba anterior, los tiempos de búsqueda del plan aumentan entre un 22 y un 29 % debido a los procesos internos del sistema operativo que penalizan al planificador.

Tras la ejecución de este nuevo banco de pruebas se comprueba que las mejoras introducidas en la capa funcional consiguen mejorar el tiempo de ejecución del plan con respecto a los bancos de pruebas 1 y 2 en torno al 23 %, una cifra que compensa el tiempo empleado en aplicar las mejoras. Además el robot es más preciso en los movimientos, lo que permite emplearlo en lugares estrechos con difícil acceso.

8.5 Cuarta prueba

Esta prueba tiene como objetivo comprobar que el robot es capaz de realizar recargas intermedias si durante la ejecución del plan no tiene batería suficiente para ejecutar el plan por completo. Para realizar esta prueba se ha creado un nuevo problema PDDL en el que se han modificado los parámetros de energía del robot, disminuyéndo la energía inicial del robot para obligarle a hacer una recarga intermedia. El escenario de la prueba sigue siendo el laboratorio E-31, en el que se realizarán 3 fotografías en distintos puntos del mapa. El análisis de los consumos del robot no se ha realizado todavía, por lo que los consumos actuales del modelo PDDL son orientativos, además un ciclo de recarga real dura entre media y una hora, por lo que al tratarse de una prueba de concepto se ha ajustado el modelo para que la recarga dure 20 segundos y así reducir el tiempo requerido para realizar la prueba. En trabajos futuros del proyecto se incluye el modelado correcto de los consumos del robot.

A partir de ahora solo se usa OPTIC$_{S\theta}$ en la realización de los experimentos, ya que para el modelado de los consumos es necesario el uso de sintáxis propia de PDDL 3, forzando la necesidad de utilizar un planificador compatible con dicha sintáxis (OPTIC$_{S\theta}$ soporta esa sintáxis, mientras que UP2TA no).

Tras la realización de la prueba el planificador obtiene el siguiente plan:

```
1  SOLUCION:
2
3  moveto turtle c6_12 c7_13 [29.2706]
4  moveto turtle c7_13 c8_15 [46.2808]
5  moveto turtle c8_15 c9_15 [20.6974]
6  moveto turtle c9_15 c10_16 [29.2706]
7  moveto turtle c10_16 c11_15 [29.2706]
8  moveptu turtle kinect p90_90 p120_100 [3]
9  takepicture turtle c11_15 kinect p120_100 highres [10]
10 moveptu turtle kinect p120_100 p90_90 [3]
11 moveto turtle c11_15 c12_16 [61.0038]
12 moveto turtle c12_16 c13_16 [43.1362]
13 autodock turtle c13_16 [20]
14 charge turtle c13_16 10.000 [20]
15 undock turtle c13_16 [11]
16 moveto turtle c13_16 c12_16 [14.1965]
17 moveto turtle c12_16 c9_16 [42.5896]
18 moveto turtle c9_16 c8_15 [20.0769]
19 moveto turtle c8_15 c8_8 [99.3758]
20 moveto turtle c8_8 c9_8 [14.1965]
21 moveto turtle c9_8 c10_8 [14.1965]
22 moveto turtle c10_8 c11_7 [20.0769]
23 moveto turtle c11_7 c13_6 [31.7444]
24 moveto turtle c13_6 c14_6 [14.1965]
25 moveptu turtle kinect p90_90 p120_100 [3]
26 takepicture turtle c14_6 kinect p120_100 highres [10]
27 moveptu turtle kinect p120_100 p90_90 [3]
28 moveto turtle c14_6 c13_6 [29.2196]
29 moveto turtle c13_6 c10_7 [92.4004]
30 moveptu turtle kinect p90_90 p120_100 [3]
31 takepicture turtle c10_7 kinect p120_100 highres [10]
```

En dicho plan se puede ver como las líneas 13, 14 y 15 contienen la información de interés, mostrando las acciones que inician el proceso de recarga, la recerga y la finalización de dicha recarga. Tras acabar

el proceso de recarga el robot continúa con la ejecución del plan. Para visualizar mejor el resultado del plan se ha representado la trayectoria seguida por el robot en la figura 8.4, siguiendo el modelo empleado en las pruebas anteriores.

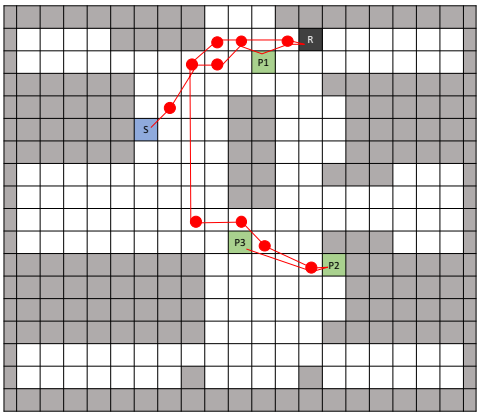


Figura 8.4: Trayectoria recorrida por el robot en el cuarta prueba

En la figura se puede apreciar la trayectoria seguida por el robot, el cuál inicia el movimiento para capturar la primera fotografía (P1). Como el planificador previamente calculó que el robot no tenía energía sufuciente para terminar el plan decide hacer una recarga intermedia en la estación de carga (R) antes de capturar las dos fotografías restantes. Cuando termina la carga el robot prosigue con el plan para capturar las dos fotografías que le quedan (P2 y P3). De esta forma se ha comprobado que el robot es capaz de forma autónoma gestionar su nivel de batería pudiendo resolver situaciones críticas en las que planes demasiado extensos no pueden ser completados con una sola carga de batería.

El tiempo de ejecución de la prueba asciende hasta los 460 segundos recorriendo una distancia total de 13,32 metros, concorde a los resultados anteriores obtenidos por OPTIC$_{S\theta}$.

8.6 Quinta prueba

Esta quinta prueba quiere comprobar que el robot es capaz de esquivar obstáculos que no aparecen en el mapa (alguien ha dejado una papelera en medio, personas, etc.). El mecanismo que permite esquivar obstáculos es el siguiente: el robot mediante sus sensores sonar es capaz de detectar obstáculos que interfieran en sus trayectoria. Cuando detecta uno de estos obstáculos se detiene el módulo de locomoción indicándole al ejecutor la casilla de detección y la orientación del robot. El ejecutor comprueba que esa casilla es errónea, por lo que añade un nuevo obstáculo en el mapa (teniendo en cuenta la posición del robot y su orientación), actualiza el problema PDDL con la nueva posición del robot e inicia un nuevo proceso de planificación en el deliberador, generando un nuevo plan. Este nuevo plan tiene en cuenta el nuevo obstáculo añadido en el mapa y la posición del robot, por lo que el juego de acciones incluídos en el nuevo plan consiguen esquivar el obstáculo. Finalmente, el nuevo plan es ejecutado en el ejecutor cumpliendo con las metas iniciales. No obstante, si los sensores de sonar fallasen, los sensores parachoque del robot entrarían en acción actúando de la misma forma que los sensores sonar para esquivar los obstáculos.

Una vez entendido el mecanismo que esquiva los obstáculos se procede a explicar el escenario sobre el que se realiza la prueba. El lugar de realización de la prueba sigue siendo el laboratorio E-31, con ligeras variantes, ya que ahora el robot inicialmente se encuentra en la estación de carga para evitar el consumo batería en estado de reposo. Al igual que en otras pruebas, se fijan 3 puntos en los que el robot tiene que

realizar fotografías, pero ahora durante el transcurso de la prueba se pone en la trayectoria del robot una papelera que debe ser detectada y esquivada, poniendo en práctica la explicación anterior.

En esta prueba, debido a la detección del nuevo obstáculo se generarán dos planes: un plan incial que cumpla los objetivos marcados inicialmente, y un segundo plan generado tras detectar el obstáculo, que esquive el obstáculo y cumpla las metas que queden por alcanzar. Los dos planes obtenidos se muestran a continuación:

```
SOLUCION 1:

undock turtle c13_16 [11]
moveto turtle c13_16 c12_16 [16.4441]
moveto turtle c12_16 c11_16 [16.4441]
moveto turtle c11_16 c9_16 [32.8881]
moveto turtle c9_16 c8_15 [23.2554]
moveto turtle c8_15 c8_8 [115.108]
moveptu turtle kinect p90_90 p90_70 [3]
takepicture turtle c8_8 kinect p90_70 highres [10]
moveptu turtle kinect p90_70 p90_90 [3]
moveto turtle c8_8 c8_9 [21.4286]
moveto turtle c8_9 c8_15 [128.571]
moveptu turtle kinect p90_90 p120_100 [3]
takepicture turtle c8_15 kinect p120_100 highres [10]
moveptu turtle kinect p120_100 p90_90 [3]
moveto turtle c8_15 c8_8 [115.108]
moveto turtle c8_8 c9_8 [16.4441]
moveto turtle c9_8 c10_7 [23.2554]
moveto turtle c10_7 c10_4 [49.3322]
moveptu turtle kinect p90_90 p90_70 [3]
takepicture turtle c10_4 kinect p90_70 highres [10]
```

```
SOLUCION 2:

moveto turtle c8_12 c7_11 [22.555]
moveto turtle c7_11 c6_10 [22.555]
moveto turtle c6_10 c7_9 [22.555]
moveto turtle c7_9 c8_9 [15.9488]
moveto turtle c8_9 c9_8 [22.555]
moveto turtle c9_8 c10_7 [22.555]
moveto turtle c10_7 c10_4 [47.8463]
moveptu turtle kinect p90_90 p90_70 [3]
takepicture turtle c10_4 kinect p90_70 highres [10]
moveptu turtle kinect p90_70 p90_90 [3]
moveto turtle c10_4 c10_7 [58.933]
moveto turtle c10_7 c9_8 [27.7813]
moveto turtle c9_8 c8_9 [27.7813]
moveto turtle c8_9 c8_8 [19.6443]
moveptu turtle kinect p90_90 p90_70 [3]
takepicture turtle c8_8 kinect p90_70 highres [10]
moveptu turtle kinect p90_70 p90_90 [3]
moveto turtle c8_8 c7_9 [27.4863]
moveto turtle c7_9 c6_10 [27.4863]
moveto turtle c6_10 c6_11 [19.4357]
moveto turtle c6_11 c6_12 [19.4357]
```

```
24 moveto turtle c6_12 c7_13 [27.4863]
25 moveto turtle c7_13 c8_15 [43.4596]
26 moveptu turtle kinect p90_90 p120_100 [3]
27 takepicture turtle c8_15 kinect p120_100 highres [10]
```

Visualizando el primer plan se observa como el robot sale de la estación de carga con la acción `undock` y comienza con la ejecucción del plan. Si todo fuese correctamente, la primera fotografía la tomaría en la casilla (8,8), sin embargo el segundo plan comienza su ejecución en la celda (8,12), lo que indica que en el trayecto desde la casilla (8,15) a la (8,8) ha aparecido un obstáculo que no estaba contemplado en el mapa. A partir de ese momento el segundo plan será el plan válido que alcance las metas establecidas. Para tener una visión gráfica de los movimientos se ha representado la trayectoria del robot en la figura 8.5.

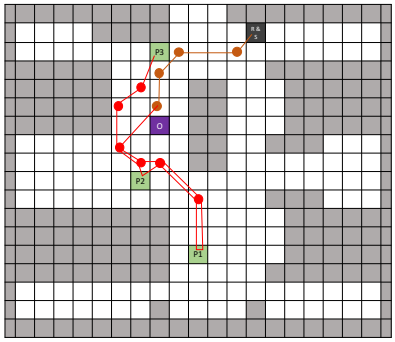


Figura 8.5: Trayectoria recorrida por el robot en la quinta prueba

En la figura se observa que la casilla de recarga tiene escrito R&S, indicando que contiene la estación de carga y que además es el punto de partida del plan. Como se ha indicado anteriormente el primer plan comienza su ejecución hasta que en la casilla (8,11) se detecta un obstáculo (casilla morada con una O) que interfiere en la trayectoria del robot (el robot avanzaría en línea recta hasta la celda (8,8)). En ese momento los sensores de sonar toman el control y detienen el módulo de locomoción, pasándole el control al ejecutor. Éste añande un nuevo obstáculo en el mapa y le indica al planificador que genere un nuevo plan en base a los datos actualizados. El robot ejecuta el nuevo plan esquivando la casilla que contiene el obstáculo para obtener las fotografías P1, P2 y P3.

La prueba tiene un tiempo de ejecucción de 434 segundos recorriendo una distancia de 15.67 metros. Aquí también es interesante señalar el tiempo que tarda el planificador en encontrar la solución al plan, cuyo valor asciende hasta los 197 ms, cifra superior al tiempo que emplea OPTIC$_{S\theta}$ en pruebas similares debido a la necesidad de realizar dos planificaciones.

8.7 Sexta prueba

En esta última prueba se pretende simular el comportamiento del robot en un escenario real (robot destinado a fines de teleasistencia en un entorno doméstico). En este entorno el robot permanecerá conectado a la base de carga siempre que esté en posición de reposo (sin ninguna misión por cumplir) para tener la batería cargada cuando le llegue una alerta. Cuando reciba dicha alerta (proveniente de un sistema externo de sensores que detecta caídas, anomalías, subida de tensión, movimientos, etc.),

el robot se dirigirá de forma autónoma a la estancia de la casa en la que se ha producido la alerta, esquivando los obstáculos que se encuentre por el camino. Además puede recibir nuevas metas por parte de un teleoperador durante la ejecución del plan con el objetivo de asegurar la integridad de la persona dependiente. Al llegar a la estancia realiza fotografías para comprobar el estado del entorno y conecta con un teleoperador que realiza las gestiones pertinentes para resolver el problema. Cuando finaliza la misión el robot vuelve a la estación de carga para estar preparado ante nuevas alertas. Para simular este escenario la prueba partirá de la base de carga y tendrá que realizar 3 fotografías en distintos puntos (que simulan 3 estancias de una casa), incluyendo además un obstáculo que el robot debe esquivar. Asímismo durante la ejecución del plan se incluirá una nueva meta que le indique que el plan finaliza en la estación de carga (simulando la inclusión de metas en medio de la ejecución del plan).

La ejecución de este nuevo escenario conlleva dos replanificaciones (una para añadir la nueva meta, y otra al esquivar el obstáculo) que añaden complejidad a la prueba y aumentan la carga del procesador. Los puntos en los que se realizan las fotografías son los mismos que los del experimento anterior, solo que ahora el obstáculo se ha movido una casilla hacia arriba. Tres serán los planes generados por estea prueba: un primer plan que contenga las 3 metas iniciales, un segundo plan en el que se añade la nueva meta e incluye las 3 metas anteriores, y el plan final que se genera para esquivar el obstáculo, cuya aparición en orden cronológico se muestra a continuación:

```
SOLUCION 1:

undock turtle c13_16 [11]
moveto turtle c13_16 c12_16 [16.4441]
moveto turtle c12_16 c11_16 [16.4441]
moveto turtle c11_16 c9_16 [32.8881]
moveto turtle c9_16 c8_15 [23.2554]
moveto turtle c8_15 c8_8 [115.108]
moveptu turtle kinect p90_90 p90_70 [3]
takepicture turtle c8_8 kinect p90_70 [10]
moveptu turtle kinect p90_70 p90_90 [3]
moveto turtle c8_8 c8_9 [21.4286]
moveto turtle c8_9 c8_15 [128.571]
moveptu turtle kinect p90_90 p120_100 [3]
takepicture turtle c8_15 kinect p120_100 [10]
moveptu turtle kinect p120_100 p90_90 [3]
moveto turtle c8_15 c8_8 [115.108]
moveto turtle c8_8 c9_8 [16.4441]
moveto turtle c9_8 c10_7 [23.2554]
moveto turtle c10_7 c10_4 [49.3322]
moveptu turtle kinect p90_90 p90_70 [3]
takepicture turtle c10_4 kinect p90_70 [10]
```

```
SOLUCION 2:

moveto turtle c11_16 c9_16 [30.1079]
moveto turtle c9_16 c8_15 [21.2895]
moveto turtle c8_15 c8_8 [105.377]
moveto turtle c8_8 c9_8 [15.0539]
moveto turtle c9_8 c10_7 [21.2895]
moveto turtle c10_7 c10_4 [45.1618]
moveptu turtle kinect p90_90 p90_70 [3]
```

```
takepicture turtle c10_4 kinect p90_70
moveptu turtle kinect p90_70 p90_90 [3]
moveto turtle c10_4 c10_7 [58.933]
moveto turtle c10_7 c9_8 [27.7813]
moveto turtle c9_8 c8_9 [27.7813]
moveto turtle c8_9 c8_8 [19.6443]
moveptu turtle kinect p90_90 p90_70 [3]
takepicture turtle c8_8 kinect p90_70 [10]
moveptu turtle kinect p90_70 p90_90 [3]
moveto turtle c8_8 c8_9 [21.4286]
moveto turtle c8_9 c8_15 [128.571]
moveptu turtle kinect p90_90 p120_100 [3]
takepicture turtle c8_15 kinect p120_100 [10]
moveptu turtle kinect p120_100 p90_90 [3]
moveto turtle c8_15 c9_15 [24.7755]
moveto turtle c9_15 c10_16 [35.0379]
moveto turtle c10_16 c13_16 [74.3266]
autodock turtle c13_16 [20]
```

```
SOLUCION 3:

moveto turtle c8_13 c7_12 [23.5145]
moveto turtle c7_12 c7_11 [16.6273]
moveto turtle c7_11 c6_10 [23.5145]
moveto turtle c6_10 c7_9 [23.5145]
moveto turtle c7_9 c8_9 [16.6273]
moveto turtle c8_9 c9_8 [23.5145]
moveto turtle c9_8 c10_7 [23.5145]
moveto turtle c10_7 c10_4 [49.8818]
moveptu turtle kinect p90_90 p90_70 [3]
takepicture turtle c10_4 kinect p90_70 [10]
moveptu turtle kinect p90_70 p90_90 [3]
moveto turtle c10_4 c10_7 [58.933]
moveto turtle c10_7 c9_8 [27.7813]
moveto turtle c9_8 c8_9 [27.7813]
moveto turtle c8_9 c8_8 [19.6443]
moveptu turtle kinect p90_90 p90_70 [3]
takepicture turtle c8_8 kinect p90_70 [10]
moveptu turtle kinect p90_70 p90_90 [3]
moveto turtle c8_8 c7_9 [27.4863]
moveto turtle c7_9 c6_10 [27.4863]
moveto turtle c6_10 c6_11 [19.4357]
moveto turtle c6_11 c6_12 [19.4357]
moveto turtle c6_12 c7_13 [27.4863]
moveto turtle c7_13 c8_15 [43.4596]
moveptu turtle kinect p90_90 p120_100 [3]
takepicture turtle c8_15 kinect p120_100 [10]
moveptu turtle kinect p120_100 p90_90 [3]
moveto turtle c8_15 c9_15 [24.7755]
moveto turtle c9_15 c10_16 [35.0379]
moveto turtle c10_16 c13_16 [74.3266]
autodock turtle c13_16 [20]
```

El primer plan es el que se genera al iniciar la prueba, en él el primer movimiento que realiza el robot es la desconexión de la estación de carga, para posteriormente moverse por el mapa con el objetivo de

realizar 3 fotografías. Sin embargo, al llegar a la casilla (11,6) le entra un nuevo objetivo (volver a la estación de carga al finalizar la misión) al planificador, por lo que tiene que replanificar, dándo lugar al segundo plan. En este segundo plan cambia el orden en el que se realiza la captura de imágenes, y aparecen nuevas órdenes que llevan al robot hasta la base de carga una vez que se han efectuado todas las fotografías. Durante la ejecución de este segundo plan aparece un obstáculo que detecta el robot, por lo que es necesario replanificar otra vez, dando lugar al tercer y último plan. Este plan continúa con la captura de imágenes y estaciona al robot en la estación de carga cunado se cumplen todas las metas. Al igual que se ha hecho en ejemplos anteriores, la figura 8.6 proporciona una imagen con la trayectoria del robot que aclara la secuencia de ejecución de la prueba.

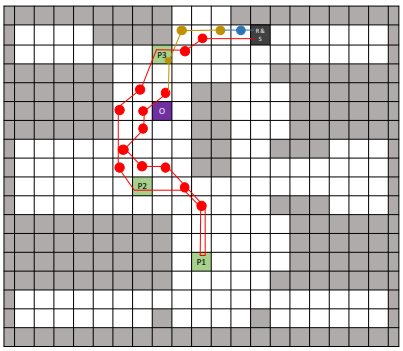


Figura 8.6: Trayectoria seguida por el robot en la sexta prueba

La figura 8.6 muestra la trayectoria seguida por el robot diferenciando los 3 planes generados. El primer plan se encuentra coloreado de color azul, el segundo de color dorado y el tercero en color rojo. Las fotografías tomadas se representan en casillas de color verde y numeradas con la siguiente nomeclatura: P1, P2 y P3, mientras que el obstáculo aparece en color morado con una O. La ejecución del plan comienza desde la base de carga (celda negra con R&S), dónde el robot se desconecta de la alimentación y comienza a moverse siguiendo la trayectoria azul. Sin embargo cuando llega a la casilla (11,6) se introduce un nuevo objetivo en el problema PDDL (el robot debe terminar el plan conectado a la base de carga) que obliga a replanificar, obteniendo así un nuevo plan. Como ya se ha indicado anteriormente este nuevo plan modifica el orden original de captura de imágenes y añade al final del mismo nuevas acciones que llevan al robot hasta la base de carga. El robot comienza la ejecución de este nuevo plan siguiendo la trayectoria de color oro, pasando por la casilla en la que se debe realizar la fotografía P3, aunque no la realiza en este momento, ya que decide dejarla para el final. La ejecución de este segundo plan se ve interrumpida por la irrupción de un obstáculo en la trayectoria del robot, que obliga a realizar una nueva planificación, obteniendo así el tercer y último plan. Este último plan queda representado en color rojo, y sus acciones llevan al robot a realizar la captura de imágenes en orden P1, P2 y P3, para finalmente estacionar el robot en la base de carga.

Con este ejemplo se ha simulado un escenario real de trabajo, ya que las 3 capturas de imágenes representan 3 posibles habitaciones de un hogar, la inclusión de una nueva meta simula nuevos objetivos que puede añadir un teleoperador durante la ejecución de una misión y el obstáculo simula elementos móviles que puede haber presentes en una casa. Además, el robot inicia y termina la misión en la base de carga, simulando el comportamiento real que tendría en una situación de reposo.

El tiempo de deliberación asciende hasta 1.071 segundos, 500 veces superior que en pruebas anteriores con el mismo planificador, debido a que en esta última prueba se han realizado dos replanificaciones más la planificación inicial. Sin embargo el tiempo de ejecución alcanza los 547, manteniéndose en la media con respecto a las pruebas anteriores.

Capítulo 9

Conclusiones y Trabajos Futuros

En este capítulo se presentan las conclusiones obtenidas basadas en un análisis técnico de las capacidades aportadas por la arquitectura de control y los resultados obtenidos en la fase de experimentación. Además se proponen futuras líneas de investigación que se deriven del trabajo que mejoren o añadan nuevos servicios a la arquitectura de control.

9.1 Conclusiones

El desarrollo de este proyecto ha tenido como objetivo demostrar y ampliar las capacidades de la arquitectura de control autónoma MoBAr desarrollada por el grupo de investigación GSI de la Universidad de Alcalá. Anterioremente esta arquitectura fue probada en simuladores y brazos robóticos, pero nunca se había integrado dentro de un robot móvil real.

La arquitectura de control autónoma MoBAr, al estar dividida en 3 niveles de abstracción, ha permitido dividir el flujo de trabajo en 3 fases bien diferenciadas (una fase por nivel). Además, esta división en niveles permite adaptar la arquitectura a varios tipos de robots, ampliando el abanico de aplicaciones a las que puede ser destinada. La capa funcional trabaja con el framework de desarrollo robótico ROS, y se encarga de realizar funciones básicas de bajo nivel interactuando con los motores y sensores del robot. Para desarrollar esta capa primero se hizo un estudio exhaustivo de dicho framework robótico con el objetivo de conocer el método de instalación, su arquitectura y varios conceptos clave. Una vez hecho el estudio se programó toda la lógica de la capa en C++, que proporciona las funciones básicas del robot (movimientos, captura de imágenes, parada de emergencia, etc.). Para dotar de mayor versatilidad a la arquitectura se integraron dos servidores RPC en la capa funcional que realizan la comunicación con el ejecutor, permitiendo una separación física de las capas de la arquitectura: la capa de bajo nivel puede ir embarcada en el robot con un hardware poco potente, mientras que las capas superiores se pueden ejecutar desde un ordenador remoto de mayor potencia.

El ejecutor permite dividir un plan con acciones de alto nivel en varias subrutinas más sencillas adaptadas a las funciones del robot, otorgando además la capacidad controlar variables de cada uno de los módulos implementados en las capas adyacentes. Para su desarrollo se ha empleado PLEXIL junto con UE, haciendo en primer lugar un estudio de ambas tecnologías. Después se implementaron los adaptadores que comunican al ejecutor con las capas adyacentes. Finalmente se desarrollaron planes de PLEXIL sencillos crecientes en complejidad hasta obtener el plan usado en la fase experimental.

El deliberador genera planes de alto nivel en base a una abstracción del entorno y del robot (dominio y problema PDDL) que no requiere un conocimiento específico sobre la estructura física del robot. Además,

el deliberador de la arquitectura mediante un archivo de configuración permite probar varios tipos de planificadores con el objetivo de obtener varias soluciones y comprobar cúal es la que obtiene el resultado óptimo. Para desarollar esta capa se tuvo que aprender la sintáxis PDDL para realizar el modelado del dominio y problema PDDL, junto al estudio del funcionamiento de la librería que controla el planificador, permitiendo su interacción y conexión con el ejecutor.

Con toda la arquitectura montada se realizaron pruebas de concepto para verificar que la arquitectura funcionaba correctamente. Además se probaron varios planificadores para ver cuál de ellos obtenía mejores resultados en términos de tiempo de ejecucción y distancia recorrida. Asímismo también se realizaron pruebas que comprobaban que el robot es capaz de recargarse de forma autónoma si detecta que su nivel de batería está por debajo de un umbral. También se probó la capacidad del robot para esquivar obstáculos que no están registrados en el mapa, simulando objetos cotidianos que pueden aparecer en una casa (sillas, papeleras, bolsas, etc.). Por último se lanzó una prueba de concepto con el objetivo de simular el comportamiento de la arquitectura en un etorno real de trabajo: una casa en la que habita una persona dependiente. En este escenario el robot habitualmente se encuentra en estado de reposo conectado a la base de carga y al detectarse una situación de peligro salta una alarma que activa el robot para que de forma autónoma vaya a inspeccionar la habitación o habitaciones en las que se originó dicha alarma. Durante esta inspección, un teleoperador externo le puede indicar al robot más metas a cumplir, asegurando la integridad de la persona dependiente. Cuando finalmente el robot cumple todas las metas vuelve a la base de carga para esperar nuevas alarmas.

Sin embargo, durante la fase de pruebas se detectaron carencias presentes en la versión actual de la arquitectura MoBAr integrada en el Turtlebot II, siendo la más importante el elevado consumo de batería en estático. Este consumo de batería viene provocado por la cámara Kinect y el mini-pc, los cuáles tienen un consumo en estático muy alto (conectado el Turtlebot II en la base de carga con todos los componentes activos, la batería del robot se descarga). Como posible solución se buscó una alternativa hardware de bajo consumo que sustituyese a estos dos componentes, encontrando en la plataforma Raspberry Pi la solución perfecta. Por ello toda la arquitectura MoBAr ha sido migrada a dicha plataforma, cuyo proceso se puede encontrar en el Anexo B.

En resumen, la arquitectura MoBAr proporciona una versatilidad enorme, ya que es posible añadir nuevas funcionalidades modificando una o varias capas. Además su diseño modular permite aumentar la complejidad del problema a resolver sin la necesidad de modificar todo el código de la arquitectura.

9.2 Trabajos futuros

En esta sección se ha realizado un análisis de los trabajos futuros que se pueden realizar sobre la plataforma robótica desarrollada en este Trabajo Fin de Máster para mejorar las capacidades de dicha arquitectura.

- En la sección experimental se ha visto como up2ta mejora los resultados en términos de tiempo de ejecucción de distancia recorrida con respecto a OPTIC$_{S\theta}$, sin embargo up2ta no soporta la sintáxis de PDDL 3 que permite entre otras cosas controlar el consumo de batería y la gestión de obstáculos. Por este motivo se propone la modificación de up2ta para que soporte la sintáxis de PDDL 3.

- La prueba realizada para comprobar la recarga autónoma del robot contenía unos consumos aproximados que no se corresponden con el consumo real del robot. Se propone medir el consumo de batería del robot en estático y cuando realiza cada acción, abriendo el circuito de alimentación del robot y midiendo sus consumos. Con esos resultados se modifica el problema PDDL indicando la

capacidad total de la batería y los consumos de cada una de las acciones. De esta forma la capa deliberativa tendría un modelo real del consumo del robot.

- En la fase de testeo se ha detectado que los sensores de ultrasonidos HC-SR04 en determinadas condiciones no detectan los obstáculos, sobre todo cuando los obstáculos presentan una superficie lisa ubicada oblicuamente con respecto al frontal del robot. Esta situación hace que el pulso de ultrasonidos que generan los HC-SR04 salga rebotado hacia delante, por lo que el sensor no recibe el eco y no visualiza el obstáculo. Para solucionarlo se podría hacer uso del sensor láser de la Kinect, pero presenta una zona de visión estrecha (58º en horizontal y 43º en vertical) y una zona muerta en los primeros 45 cm, además de la imposibilidad de detectar cristales y superficies pulidas que impiden su utilización para detectar obstáculos con precisión. Por este motivo se cree necesario añadir unos sensores de infrarrojos que sustituyan a los de ultrasonidos y que mejoren la capacidad de detección de obstáculos. El modelo recomendado son los sensores SHARP GP2Y0A21YK cuyo rango de detección va desde los 10 a los 80 cm. Además estos sensores ya han sido integrados en la base Kobuki [45], dónde se puede ver todas las características técnicas de los mismos y soportes en 3D para incorporarlos a dicha base.

- La limitación de presupuesto ha hecho que la construcción de la *pantilt* se haya realizado con piezas de aeromodelismo sobrantes de otros proyectos. Esta construcción casera afecta en la precisión de la *pantilt*, ya que tiene ligeras holguras que añaden vibraciones innecesarias. Para resolver el problema se pueden comprar *pantilts* fabricadas que proporcionan un alto rendimiento y un posicionamiento ajustado para altas cargas, o bien realizar un diseño con una impresora 3D que genere piezas a medida cuyo ensamblaje no genere holguras.

- La medida odométrica con los sensores del robot comete pequeños errores al realizar la medición que con el paso del tiempo se acumulan dando lugar a estimaciones de la posición incorrectas. Por este motivo es necesario incluir algún sistema de posicionamiento externo que fije la posición del robot eliminando los errores de odometría, pudiendo emplearse etiquetas RFID, posicionamiento inalámbrico con redes wifi o bluetooth o LED infrarrojos, por ejemplo.

- La realización de mapas para la arquitectura de control MoBAr en la actualidad es demasiado precaria, ya que hay que realizar manualmente los mapas del entorno por el que moverá el robot. Por este motivo se proporne la creación de mapas haciendo uso de técnicas de SLAM, generando así mapas automáticamente de las viviendas en las que se despliegue el robot. ROS proporciona un paquete de navegación que es capaz de crear mapas con los datos de odometría y los datos proporcionados por la Kinect, pudiéndose utilizar dicho paquete para generar los mapas adaptándolos a la estructura de mapas que tiene la capa deliberativa.

- Sustitución del robot actual por uno de menor tamaño que tenga mayor movilidad en interiores. Dado que la arquitectura MoBAr es una arquitectura modular, solamente sería necesario modificar las funciones de bajo nivel implementadas en la capa funcional.

Capítulo 10

Presupuesto

En este capítulo se detalla el presupuesto de ejecución del proyecto desglosando el coste de material, coste de licencias software, mano de obra y otros costes asociados al proyecto.

10.1 Coste de material

En este apartado se extrae el coste de los elementos hardware utilizados en el proyecto.

Concepto	Cantidad	Coste Unitario	Total
TurtleBot II Base Kobuki Batería baja capacidad (4S1P) Cargador Cable USB Plataformas y soportes	2	536,36 €	1.072,72 €
Estación de carga	2	37,19 €	74,38€
Batería alta capacidad (4S2P)	2	57,85 €	115,70 €
Microsoft Kinect	1	69,84 €	69,84 €
Soporte metálico pantilt	8	4,25 €	34,00 €
Servomotor MG996R	4	7,91 €	31,64 €
Arduino UNO R3 ATmega328	2	19,79 €	39,59 €
Sensores HC-SR04	2	2,49 €	4,98 €
Conectores y cables	1	19,89 €	19,89 €
Mini-pc	1	299,99 €	299,99 €
Raspberry Pi con accesorios Raspberry Pi Cargador Cámara Disipadores Caja	1	70,00 €	70,00 €
Ordenador portatil	1	700,00 €	700,00 €
		TOTAL:	2.532,73 €
		I.V.A.	21 %
		TOTAL CON I.V.A.	**3.064,60 €**

10.2 Coste de licencias

Este apartado detalla el coste de licencias software asociadas a los sistemas operativos, programas y frameworks de desarrollo utilizados en el proyecto. El conjunto de software empleado para el desarrollo del proyecto es el siguiente:

- Distribución Ubuntu 14.04.
- Distribución de Raspbian.
- Frameworks de ROS y PLEXIL.
- IDE Arduino.
- Editor de textos *gedit*.
- Compiladores *gcc* y *make*.
- TextMaker.

Todo el software empleado tiene licencia de código libre, por lo que no conlleva ningún coste asociado, siendo el coste total de licencias software de 0 €.

10.3 Mano de obra y otros gastos

En este apartado se desglosa el coste de personal asociado al proyecto en base al Convenio de Empresas de Ingeniería y Oficinas de Estudios Técnicos y los gastos generales asociados al proyecto.

- **Mano de obra:** En el convenio queda establecido que la retribución anual correspondiente al Nivel 1 (Licenciados y titulados 2º y 3º ciclo y Analistas) asciende a 23.618,28 €. El tiempo de ejecución del proyecto ha sido de 21 meses a tiempo parcial, por lo que el coste asociado a mano de obra asciende a 20.665,99 €.
- **Gastos generales:** Aquí se engloban los costes asociados a material de oficina, mantenimiento, desgate de herramientas, electricidad, climatización, etc., estimándose su coste un 10 % del coste de material y mano de obra. El coste final de los gastos generales asciende a 2.203,39 €.
- **Redación y dirección:** Los gastos de redacción y dirección representan un 8 % del coste de material y mano de obra, ascendiendo a 3.084,71 €.

Concepto	Coste
Mano de obra	20.665,99 €
Gastos generales	2.373,06 €
Redacción y dirección	1.898,45 €
TOTAL	24.937,50 €
I.V.A.	21 %
TOTAL CON I.V.A.	**30.174,37 €**

10.4 Presupuesto final

Finalmente el coste total del proyecto con I.V.A. incluído asciende a:

Concepto	Coste
Material Hardware	3.064,60 €
Licencias software	0.00 €
Mano de obra y otros gastos	30.174,37 €
TOTAL (I.V.A incluido)	**33.238,97 €**

En Alcalá de Henares, a 29 de septiembre de 2016

Fdo: Diego López Pajares

Apéndice A

Ampliación sobre ROS

Este anexo pretende ampliar los conocimientos adquiridos sobre el sistema operativo robótico ROS centrándose en su parte práctica (instalación, herramientas, simuladores). El capítulo comenzará haciendo una recopilación de los comandos necesarios para realizar una instalación limpia de ROS junto con las ódenes básicas que añaden los paquetes específicos del TurtleBot II. La segunda sección muestra como se crean los directorios de trabajo y los pasos a seguir para crear nuevos paquetes. La tercera parte del capítulo se centra en herramientas de bajo nivel imprescindibles para hacer un uso eficiente de ROS, analizándose en el último apartado algunas de las herramientas gráficas y simuladores robóticos integrados en ROS.

A.1 Instalación de ROS

En esta parte del anexo se recogen los pasos que se deben seguir para realizar una instalación limpia de ROS Indigo en un PC con sistema operativo Ubuntu 14.04. En la Wiki de ROS [46] y en la Wiki del TurtleBot II [47] se puede encontrar el proceso completo de instalación, pero para simplificarlo se han decidido agrupar todas las órdenes de instalación en este apartado. Se va a dividir el proceso de instalación en dos partes: una primera parte que consiste en la instalación de ROS Indigo, y una segunda parte en la que se añaden los paquetes específicos del TurtleBot II.

Para realizar la primera parte de la instalación (ROS Indigo) es necesario abrir una terminal y ejecutar en ella los siguientes comandos:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu trusty main" > /etc/apt/sources.list.d/ros-latest.list'
wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -O - | sudo apt-key add -
sudo apt-get update
sudo apt-get install ros-indigo-desktop-full
sudo rosdep init
rosdep update
echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
source ~/.bashrc
sudo apt-get install python-rosinstall
```

Una vez que se ha instalado la versión básica de ROS, se debe comenzar la segunda parte de la instalación (paquetes específicos del TurtleBot II). Esta instalación se realiza igual que la primera, abriendo

una terminal e introduciendo en ella las siguientes instrucciones:

```
sudo apt-get install ros-indigo-turtlebot ros-indigo-turtlebot-apps ros-indigo-turtlebot-interactions ros-indigo-turtlebot-simulator ros-indigo-kobuki-ftdi
. /opt/ros/indigo/setup.bash
rosrun kobuki_ftdi create_udev_rules
echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
mkdir ~/rocon
cd ~/rocon
wstool init -j5 src https://raw.github.com/robotics-in-concert/rocon/indigo/rocon.rosinstall
source /opt/ros/indigo/setup.bash
rosdep install --from-paths src -i -y
catkin_make
mkdir ~/kobuki
cd ~/kobuki
wstool init src -j5 https://raw.github.com/yujinrobot/yujin_tools/master/rosinstalls/indigo/kobuki.rosinstall
source ~/rocon/devel/setup.bash
rosdep install --from-paths src -i -y
catkin_make
mkdir ~/turtlebot
cd ~/turtlebot
wstool init src -j5 https://raw.github.com/yujinrobot/yujin_tools/master/rosinstalls/indigo/turtlebot.rosinstall
source ~/kobuki/devel/setup.bash
rosdep install --from-paths src -i -y
catkin_make
```

Con la ejecucción de todos estos comandos daría por finalizada la instalción de ROS dejando el sistema operativo robótilo listo para comenzar a trabajar.

A.2 Espacio de trabajo

Una vez realizada la instación de ROS es necesario crear un directorio de trabajo que albergue los nuevos paquetes. Para obtener dicho espacio de trabajo no basta con crear una carpeta y guardar allí los archivos de código fuente, sino que es necesario inicializar el directorio de trabajo para que ROS tenga costancia de los nuevos paquetes creados y así poder localizarlos con rapidez a través de sus herramientas de bajo nivel

A.2.1 Creación del espacio de trabajo

ROS proporciona la herramienta `catkin` para generar e inicializar de forma automática los directorios de trabajo. La creación de un directorio de trabajo se realiza ejecutando los siguientes comandos en una terminal:

```
mkdir -p ~/myworkspace/src
cd ~/myworspace/src
```

```
catkin_init_workspace
cd ..
catkin_make
```

Con estos pasos se crearía el directorio de trabajo `/myworkspace`. Al ejecutar la orden `catkin_make` se crean dos nuevas carpetas dentro del directorio de trabajo `/myworkspace`: la carpeta `/build` y la carpeta `/devel`, necesarias para el sistema operativo robótico. Sin embargo todavía ROS no es capaz de encontrar la ruta del nuevo directorio de trabajo, cuya solución pasa por añadir en el archivo `.bashrc` del sistema operativo Ubuntu una variable de entorno que localice el directorio de trabajo. Para añadir dicha variable de entorno hay que abrir el archivo `.bashrc` con un editor de texto y añadir al final del archivo la siguiente línea de código:

```
source ~/myworkspace/devel/setup.bash
```

Una vez añadido el comando al archivo, ROS encontrará cualquier paquete contenido en el directorio `/myworkspace`.

A.2.2 Creación de nuevos paquetes

Con el directorio de trabajo creado ya se pueden generar nuevos paquetes que añadan funcionalidades desarrolladas por el usuario a los robots. Al igual para crear el directorio de trabajo es necesaria la herramienta catkin para generar los nuevos paquetes. Los pasos a seguir se describen a continuación:

```
cd ~/myworkspace/src
catkin_create_pkg <nombre_nuevo_paquete> <dependencia_1> <dependencia_2> ... <dependencia_n>
cd ..
catkin_make
```

En primer lugar lugar hay que ubicarse en la carpeta `/src` del directorio de trabajo para indicarle a la herramienta `catkin` el nombre del nuevo paquete y sus dependencias. Por último se sube hasta el directorio de trabajo para que `catkin` compile el nuevo paquete. Tras ejecutar estas ódenes en la carpeta `/myworkspace/src` aparecerá una nueva carpeta con el nombre del nuevo paquete. En esa nueva carpeta aparecerá otro directorio `/src` en el que se han de ubicar los archivos de código fuente.

A.3 Herramientas de bajo nivel

ROS proporciona un conjunto de herramientas que permiten moverse de forma ágil para buscar información sobre nodos, topics, mensajes, servicios, etc. Estas herramientas se manejan mediante línea de comandos con ayuda de una terminal y son un elemento esencial para cualquier persona que trabaje con ROS. Además gracias a estas herramientas se puede hacer depuración en tiempo de ejecucción para verificar que la red funciona adecuadamente, ya que puede obtener prácticamente toda la información de la red de ROS por consola. En los siguientes apartados se irán enumerando cada una de las aplicaciones de bajo nivel que tiene implementadas por defecto ROS.

A.3.1 Rospack

Rospack es la herramienta encargada de la administración de los paquetes en ROS. Con ella se puede buscar un determinado paquete, ver sus dependencias así como ver los paquetes que dependen de un paquete. Para ejecutar esta herramienta basta con abrir una terminal y ejecutar los comandos listados en la tabla A.1.

Tabla A.1: Resumen de comandos para rospack

Comando	Función
rospack find *<paquete>*	Devuelve la ruta en la que se encuentra el paquete
rospack depends *<paquete>*	Imprime las depencias del paquete
rospack depends-on *<paquete>*	Devuelve la lista de paquetes que dependen del paquete introducido

A.3.2 Roscore

Roscore es el programa que arranca el maestro de ROS junto con unos programas prerequisitos que inician la red de ROS. Es el primer elemento que se debe arrancar antes de que se inicie ningún nodo, ya que sin él las comunicaciones entre nodos no son posibles. Para ejecutarlo unicamente hay que abrir una terminal y ejecutar:

```
roscore
```

Tras ejecutar el comando arranca el maestro y la red de ROS está lista para trabajar.

A.3.3 Rosrun

Rosrun permite ejecutar un nodo ejecutable de cualquier paquete sin tener que ir a la ruta que contiene dicho ejecutable. Para arrancar el ejecutable basta con abrir una terminal y ejecutar:

```
rosrun <paquete> <nombre_del_ejecutable>
```

A.3.4 Roslaunch

Roslaunch permite la ejecucción de varios nodos de forma simultánea con tan solo ejecutar un comando. También se pueden asignar valores por defecto al servidor de parámetros o permitir que un nodo arranque de nuevo ante un cierre inesperado. Todas estas opciones se configuran en un archivo `.launch` con sintáxis XML. Para ejecutar el archivo con extensión `.launch` basta con ejecutar el siguiente comando en una terminal:

```
roslaunch <paquete> <nombre_del_archivo.launch>
```

A.3.5 Rosnode

Rosnode es la herramienta que interactua con los nodos activos de la red. Ofrece varias posibilidades, entre las que destacan la capacidad de matar nodos y la capacidad de realizar test de conexión para verificar que un nodo está vivo. Para ejecutar rosrun hay que abrir una terminal y ejecutar en ella alguna de las acciones recogidas en la tabla A.2.

Tabla A.2: Resumen de comandos para rosnode

Comando	Función
rosnode info *<nodo>*	Imprime información sobre el nodo
rosnode kill *<nodo>*	Mata un nodo en ejecucción
rosnode list	Imprime un listado con los nodos activos
rosnode machine *<hostname>*	Imprime los nodos en ejecucción de una máquina en concreto
rosnode ping *<nodo>*	Verifica que hay conectividad con el nodo
rosnode cleanup	Limpia el registro de los nodos inalcanzables

A.3.6 Rostopic

Rostopic es la herramienta que permite interaccionar con los topics que hay activos en la red, pudiendo conocer la información que pasa por el topic, el tipo de datos que tiene el topic, su frecuencia de publicación, etc. Rostopic se ejecuta introduciendo los comandos de la tabla A.3 en una terminal.

Tabla A.3: Resumen de comandos para rostopic

Comando	Función
rostopic bw *<topic>*	Muestra el ancho de banda que usa el topic
rostopic echo *<topic>*	Imprime los mensajes que pasan por el topic
rostopic find *<msg-type>*	Devuelve los topic que están utilizando dicho tipo de mensaje
rostopic hz *<topic>*	Retorna la frecuencia de publicación de un topic
rostopic info *<topic>*	Imprime información sobre el topic
rostopic list	Lista todos los topics activos
rostopic pub *<topic><msg-type>[datos]*	Publica en el topic los datos especificados
rostopic type *<topic>*	Muestra el tipo de mensaje que transporta el topic

A.3.7 Rosmsg

Rosmsg se emplea para trabajar con los mensajes de ROS, obteniedo información sobre los mismos así como todos los mensajes que contiene un paquete o todos los mensajes de ROS. La ejecucción de la herramienta se hace a través de una consola, introduciendo en ella los comandos de la tabla A.4.

Tabla A.4: Resumen de comandos para `rosmsg`

Comando	Función
`rosmsg show <msg-type>`	Imprime los campos del mensaje
`rosmsg list`	Lista todos los mensajes existentes
`rosmsg package <paquete>`	Devuelve todos los mensajes de un paquete
`rosmsg packages`	Retorna todos los paquetes que contienen algún mensaje
`rosmsg md5 <msg-type>`	Devuelve la suma md5 del mensaje

A.3.8 `Rosservice`

Al igual que para los mensajes, ROS también proporciona la herramienta `rosservice` que interactúa con los servicios, pudiendo realizar llamadas a los mismos así como conocer todos los servicios existentes en el sistema. La ejecucción de la herramienta se realiza introduciendo los comandos de la tabla A.5 en un terminal.

Tabla A.5: Resumen de comandos para `rosservice`

Comando	Función
`rosservice args <srv>`	Devuelve los argumentos que necesita el servicio
`rosservice call <srv>[argumentos]`	Hace una llamada al servicio
`rosservice find <srv-type>`	Muestra los servicios del tipo especificado
`rosservice list`	Lista todos los servicios
`rosservice info <srv>`	Retorna información sobre el servicio
`rosservice node <srv>`	Muestra el nodo que ofrece dicho servicio
`rosservice type <srv>`	Devuelve el tipo de servicio
`rosservice uri <srv>`	Retorna la dirección URI del servicio

A.3.9 `Rosparam`

Para poder establecer una comunicación además de conocer sus parámetros de configuración, ROS proporciona la herramienta `rosparam`. Para ejecutar esta herramienta hay que introducir los comandos de la tabla A.6.

Tabla A.6: Resumen de comandos para `rosparam`

Comando	Función
`rosparam list`	Lista los nombres de los parámetros activos
`rosparam get <parametro> list`	Devuelve el valor del parámetro
`rosparam set <parametro>[valor] list`	Fija el valor especificado en el parámetro especificado
`rosparam delete <parametro> list`	Elimina el parámetro
`rosparam dump <archivo> list`	Guarda todo el contenido del servidor de parámetros en un archivo `.yaml`
`rosparam load <archivo> list`	Carga el contenido de un fichero `.yaml` al servidor de parámetros

Rosparam también hace uso de la terminal para su ejecucción, siguiendo el procedimiento visto en las herramientas anteriores.

A.4 Herramientas de simulación y visualización

ROS proporciona un set de herramientas de desarrollo muy potentes que permiten obtener de forma gráfica la interconexión de todos los nodos de la red, niveles de sensores, y modelos de simulación entre otras cosas. A continuación se van a ver algunas de las herramientas que permiten interactúar de forma gráfica con ROS.

A.4.1 ROS GUI

Es la interfaz gráfica de usuario de ROS que permite interactuar de forma visual con todo el framework de desarrollo robótico. Además, al estar basada en un sistema de plugins, permite desarrollar nuestras propias herramientas así como reutilizar otras que ya estén implementadas. Como ROS GUI contiene numerosos plugins, únicamente se van a ver aquí los más utilizados por la comunidad.

`rqt_graph`

`rqt_graph` es una herramienta que permite visualizar el grafo de la red de ROS de forma similiar a las representaciones gráficas de nodos topics y servicios, realizadas en el apartado 5.2. Con esta herramienta se pueden conocer las dependencias entre nodos que exiten en la red, así como sus conexiones. Para ejecutar dicha herramienta hay que abrir una consola y ejecutar el siguiente comando:

```
rosrun rqt_graph rqt_graph
```

Al ejecutar dicho comando con todos los paquetes de TurtleBot II lanzados en la red de ROS, aparece el grafo de la figura A.1a. En dicha representación gráfica, los nodos quedan representados como óvalos, mientras que los topics son representados como cuadrados.

`rqt_robot_monitor`

La herramienta `rqt_robot_monitor` hace uso del topic de diagnostic para obtener un feedback del estado del robot y sus sensores. La aplicacion recoge toda esta información junto a su nivel de error (OK, WARNIG y ERROR) y la muestra de forma gráfica en una ventana en la que se pueden ir expandiendo todos los apartados. La figura A.1b muestra el diagnóstico del TurtleBot II utilizado en el proyecto. En la imagen se puede ver como aparece un error en el sistema de energía, debido a que los drivers de la batería del ordenador con los que se realizó la captura de imagen no son compatibles con la versión de ROS utilizada. La ejecución de esta herramienta se realiza introduciendo un una terminal el siguiente comando:

```
rosrun rqt_robot_monitor rqt_robot_monitor
```

rqt_topic

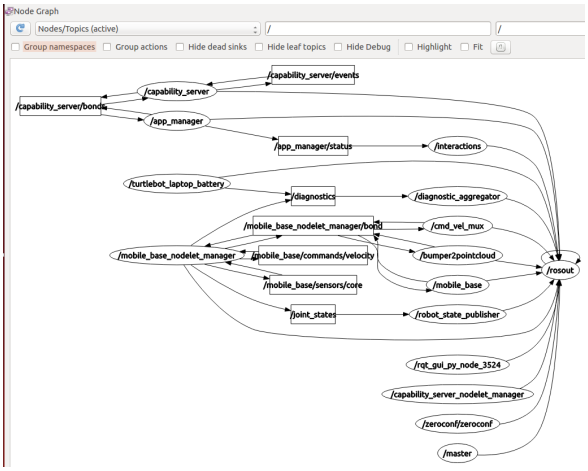
Rqt_topic permite visualizar de forma gráfica la información de todos los topic activos en la red. Esta herramienta agiliza el proceso de visualización de datos frente a la herramienta complementaria en consola (rostopic) debido a la rapidez en la selección visual de topics y la organización de los datos mostrados por pantalla. Para tener una visión gráfica de la herramienta se proporciona la figura A.1c, cuyo lanzamiento se realiza ejecutando el siguiente comando en la terminal:

```
1 rosrun rqt_topic rqt_topic
```

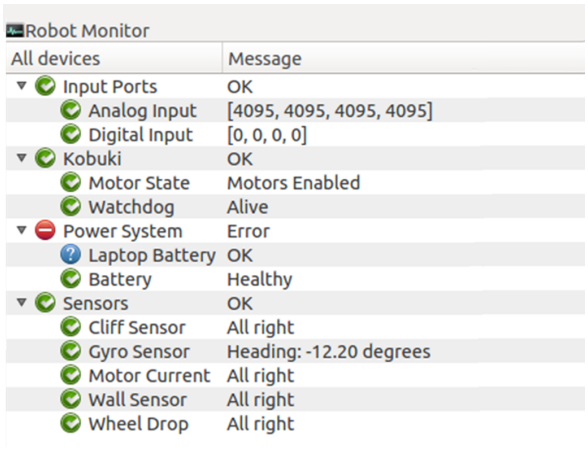
rqt_plot

La herramienta rqt_robot_plot (figura A.1d) permite representar de forma gráfica en una ventana 2D los datos de cualquier sensor en función del tiempo. Esta herramienta permite añadir los datos de sensores que se deseen, añadiendo para cada uno de ellos una traza nueva. La ejecucción de esta herramienta se realiza ejecutando el siguiente comando en la terminal:

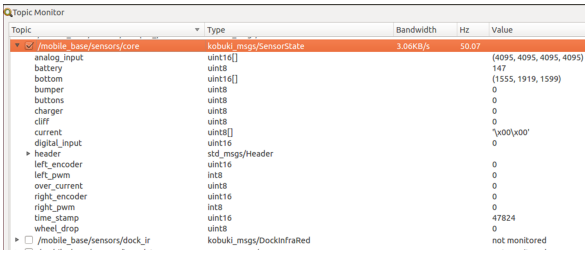
```
1 rosrun rqt_plot rqt_plot
```



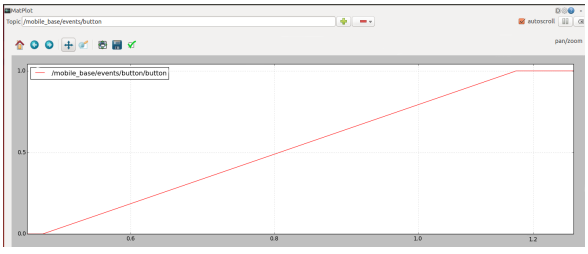
(a) Herramienta rqt_graph



(b) Herramienta rqt_robot_monitor



(c) Herramienta rqt_topic



(d) Herramienta rqt_plot

Figura A.1: Plugins de ROS GUI

A.4.2 Rviz

Rviz es un visualizador en tres dimensiones que muestra información sobre el estado de ROS y sobre los datos de los sensores del robot. Con esta herramienta es posible tener una imagen en 3D del robot junto con la información que recogen sus sensores. Esta herramienta es de gran utilidad ya que es posible mover el robot físicamente por un entrono real y tener una representación del mismo en la pantalla del ordenador, mostrando en tiempo real la información de todos sus sensores. Además incluye aplicaciones para cargar un mapa y hacer navegación autónoma de forma gráfica, indicando los objetivos mediante el uso del ratón.

La figura A.2, muestra la representación del robot utilizado en el proyecto dentro de la herramienta Rviz. La herramienta permite capturar el streaming de vídeo que recoge la Kinect en vivo (zona inferior izquierda), a la vez que hace una representación gráfica de los datos obtenidos por el sensor de profundidad que lleva incorporados la Kinect (zona derecha). Junto a estos datos la herramienta incluye un modelo visual del robot que incluye una representación gráfica de los datos recogidos por los sensores, facilitando así la comprensión de los datos por parte del usuario. Este modelo del robot al tratarse de un robot comercial ya viene integrado dentro de los paquetes de instalación del TurtleBot, no obstante si no se dispone del modelo del robot, es posible crearlo manualmente.

Finalmente para lanzar aplicación hay que ejecutar en el terminal el siguiente comando:

```
1 roslaunch turtlebot_rviz_launchers view_robot.launch
```

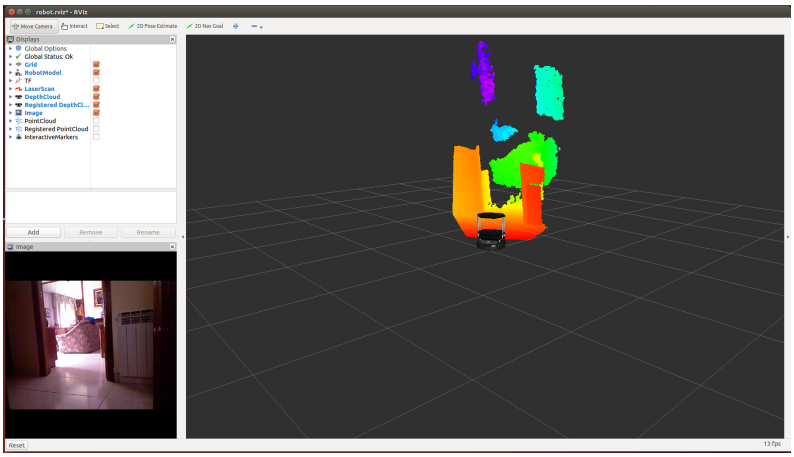


Figura A.2: Herramienta Rviz

A.4.3 Gazebo

Gazebo es un simulador multirobot para entornos complejos que tiene la capacidad de simular numerosos robots, sensores y objetos del mundo real en 3D. Es capaz de generar un comportamiento real de los robots y su interacción con los objetos que lo rodean. Normalmente Gazebo se utiliza para diseñar robots, testear algortimos o realizar test de regresión es escenarios reales, debido a su potencial y a la gran versatilidad que ofrece creando mundos virtuales de forma sencilla con herramientas CAD.

En cuanto su integración con ROS, Gazebo es independiente de ROS y se puede instalar como un paquete habitual de Ubuntu, aunque está completamente integrado con ROS gracias a plugins que hacen una integración perfecta entre los dos sistemas.

Al igual que ocurría en la sección anterior, al trabajar con un robot comercial se encuentra implementado un modelo visual y mecánico para simulación. Para trabajar con la plataforma TurtleBot II en Gazebo es necesario ejecutar el siguiente comando:

```
roslaunch turtlebot_gazebo turtlebot_world.launch
```

Tras ejecutar el comando se abre una aplicación cuya resultado queda recogido en la figura A.3. Al igual que ocurría con el robot real, es posible que Rviz muestre toda la información de los sensores simulados del robot cuando se utiliza el simulador Gazebo, cuya muestra queda recogida en la figura A.4

Como se puede apreciar en la figura A.4, Rviz muestra a través de los sensores simulados de profundidad el cubo que aparece en el mundo Gazebo de la figura A.3. Además se puede ver como Rviz representa fielmente dicho mundo ya que también queda reprensentado el vídeo de la Kinect en la esquina inferior izquierda.

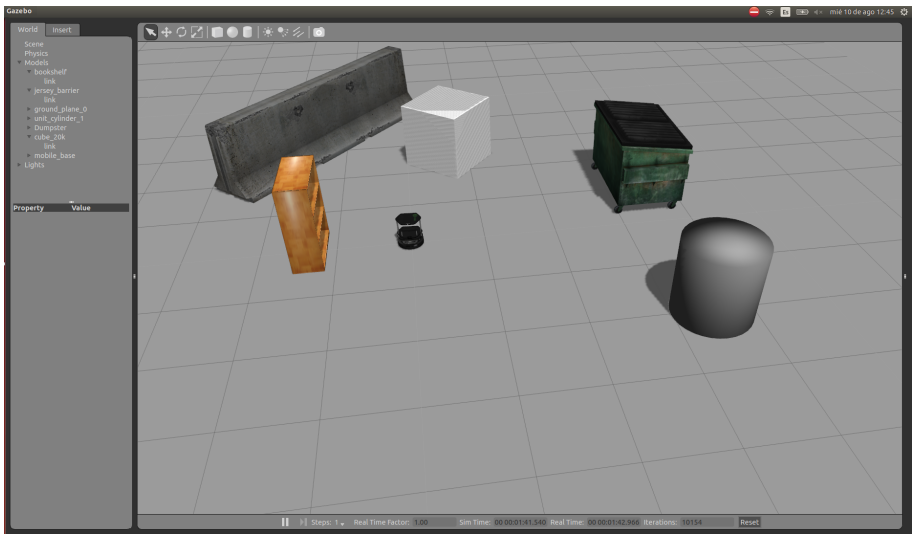


Figura A.3: Gazebo integrado con TurtleBot II

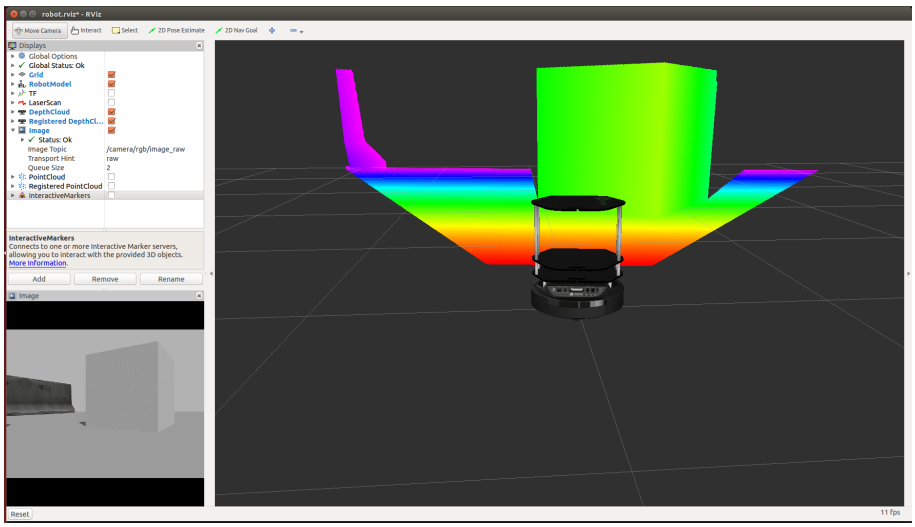


Figura A.4: Rviz mostrando los datos de Gazebo

Apéndice B

Integración de MOBAR en una arquitectura hardware de bajo consumo

Este anexo tiene como objetivo mostrar la integración de la arquitectura MoBAr en una arquitectura hardware de bajo consumo y bajo coste. Primero se hace una introducción a las plataformas de bajo coste, encuadrando su evolución y se muestra a opción escogida. En la sección B.3 se explica el proceso de migración de la arquitectura MoBAr a la plataforma Raspberry Pi. Finalmente se muestra la versión del TurtleBot II con la aquitectura MoBAr integrada en una Raspberry Pi.

B.1 Introducción

Los hábitos de la sociedad de consumo están cambiando debido a la eliminación de fronteras entre países junto con el auge de la sociedad de la información. Estos cambios han provocado que el concepto de ordenador personal que se tenía hasta el momento haya cambiado, pasando de ordenadores fijos ubicados en una estancia de la casa, a toda una gama de dispositivos portátiles con rendimiento elevado y autonomías impensables hasta la fecha. En esta evolución ha tenido mucho que ver el desarrollo tecnológico experimentado en la última década, disminuyendo el tamaño de los dispositivos electrónicos y reduciendo su en gran medida su consumo. Es por eso que hoy en día se pueden encontrar móviles, tablets, placas de desarrollo con autonomías superiores al día y con una capacidad de procesamiento superior al de los ordenadores de años atrás. La empresa ARM ha tenido mucho que ver en esta evolución tecnológica, ya que ha desarrollado una arquitectura Reduced Instruction Set Computer (RISC) ideal para aplicaciones de bajo consumo. De hecho el diseño de sus microprocesadores está implementado en numerosos dispositivos (móviles, tablets, robots, ordenadores de bajo coste, etc.), siendo un referente en el mercado actual y cuyo futuro se espera realmente prometedor.

La idea de utilizar una arquitectura ARM de bajo consumo sobre la que montar la arquitectura MoBAr es ideal, ya que gracias a su consumo reducido aumentaría notablemente la autonomía del robot. Entre todas las alternativas existentes en el mercado se escogió la plataforma Raspberry Pi debido a su bajo coste, su integración con ROS y su soporte con sistemas operativos Linux. En la siguiente sección se conocerá con un poco más de detalle esta plataforma.

B.2 Raspberry Pi

Raspberry Pi es un ordenador de bajo coste desarrollado por la Fundación Raspberry Pi con el objetivo de estimular la educación informática en países subsdesarrollados. Este ordenador posibilita la creación de pequeños proyectos hardware incrementando las capacidades de los microcontroladores tradicionales puesto que permiten instalar un sistema operativo Linux, y además poseen conexiones GPIO que permiten interaccionar con diferentes componentes electrónicos (sensores, actuadores, etc.)

En cuanto al hardware, actualemnte se pueden encontrar 4 versiones de Raspberry: modelo 1A, 1B+, 2B, 3B y Zero, cuyas especificaciones técnicas se encuentran en la tabla B.1.

Tabla B.1: Especificaciones técnicas Raspberry Pi

	Raspberry Pi 1 model A	**Raspberry Pi 1 model B+**	**Raspberry Pi 2 model B**	**Raspberry Pi 3 model B**	**Raspberry Pi Zero model**
CPU	ARM11 700 Mhz	ARM11 700 MHz	ARM Cortex A7 900 MHz (quad-core)	ARMv8 1.2 GHz (quad-core)	ARMv11 1GHz
RAM	256 MB	512 MB	1GB	1GB	512MB
USB	1	4	4	4	2
Red	-	10/100 Ethernet	10/100 Ethernet	10/100 Ethernet, Wifi, Bluetooth	-
Almacenaje	SD/MMC	MicroSD	MicroSD	MicroSD	Micro sD
Perifericos	40 GPIO	40 GPIO	40 GIPO	40 GPIO	40 GPIO
Consumo	500 mA	600 mA	800 mA	800 mA	268mA

La primera versión de Raspberry Pi (1 model A) poseía un hardware muy limitado, ya que tan solo contaba con 256 MB de RAM y un procesador mononúcleo con una frecuencia de 700 MHz. Sin embargo pronto aparecieron nuevos modelos que incrementaban las especificaciones hardware, aumentando así el rendimiento. El modelo utilizado para realizar la migración es una Raspberry Pi 2 model B, la primera que incorporó un procesador de 4 núeclos y 1 GB de memoria RAM debido a que en el momento de realizar la migración era la última placa en salir al mercado.

B.3 Proceso de migración

En esta sección se muestra el proceso de migración de la arquitectura MoBAr a la plataforma de bajo coste Raspberry Pi. Se comienza con la instalación del sistema operativo y los frameworks de desarrollo necesarios para MoBAr, pasando a las modificaciones estructurales efectuadas en el robot y finalizando con la migración del código fuente.

B.3.1 Instalación de requisitos

Para realizar la migración de la arquitectura MoBAr previamente hay que instalar un sistema operativo Linux soportado por Raspberry que sea compatible con ROS. Este sistema operativo es Raspbian[1] [48],

[1] A partir del modelo 2 de Raspberry Pi es posible instalar otros sistemas operativos en dicha plataforma, como por ejemplo Windows 10 IoT Core, o Ubuntu Mate.

un sistema operativo libre basado en Debian y optimizado para la Raspberry Pi que además contiene un set básico de programas y utilidades para comenzar a trabajar. El proceso de instalación es sencillo cuyo comienzo pasa por descargarse la imagen del sistema operativo desde la página web oficial de Raspberry [49]. Una vez que se ha descargado la imagen hay que descomprimirla en una carpeta y hacer una copia binaria en una tarjeta microSD. La copia de la imagen binaria se hace ejecutando el siguiente comando en una terminal:

```
dd bs=4M if = directorio_que_contiene_imagen.img of=/dev/mi_tarjetaSD
```

Una vez realizada la copia, se inserta la tarjeta microSD la placa Raspberry Pi, se enchufa un teclado al puerto USB, se conecta a una pantalla a través de un cable HDMI y se alimenta, iniciándose el proceso de instalación. Este proceso de instalación consiste en un método guiado de configuraciones básicas que el usuario ajusta a sus necesidades (expansión del sistema de ficheros, cambio de usuario y contraseña, cambio de hora, configuración de la tarjeta de red, etc.). Al finalizar el proceso de instalación la Raspberry Pi está lista para trabajar, pudiendo pasar al siguiente punto de la instalación: la instalación de los frameworks de desarrollo necesarios en MoBAr (ROS y PLEXIL).

La instalación de ROS en Raspbian puede resultar tediosa debido a problemas con paquetes ROS que no tienen soporte en Raspbian, teniendo que instalar los mismos a mano. De cualquier forma los pasos básicos de instalación de ROS se describen a continuación:

En primer lugar es necesario añadir y actualizar los repositorios en los que se encuentra la distribuciónde ROS para Raspbian, ejecutando las siguientes líneas de código en una terminal.

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu wheezy main" > /etc/apt/sources.
    list.d/ros-latest.list'
$ wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -O - | sudo apt-key add
    -
$ sudo apt-get update
$ sudo apt-get upgrade
```

El segundo paso consiste en instalar las herramientas necesarias para compilar e instalar las dependencias de ROS:

```
$ sudo apt-get install python-pip python-setuptools python-yaml python-argparse python-
    distribute python-docutils python-dateutil python-six
$ sudo pip install rosdep rosinstall_generator wstool rosinstall
$ sudo rosdep init
$ rosdep update
```

Por último se instala ROS:

```
$ mkdir ~/ros_catkin_ws
$ cd ~/ros_catkin_ws
$ rosinstall_generator ros_comm --rosdistro indigo --deps --wet-only --exclude roslisp --tar >
    indigo-ros_comm-wet.rosinstall
$ wstool init src indigo-ros_comm-wet.rosinstall
```

Con esto se tendría una instalación básica de ROS en Raspbian, sin embargo los paquetes específicos del TurtleBot II no están incluidos. Para su instalación se siguen los pasos de la segunda parte del anexo A.1, obteniendo así tras su instalación la versión completa de ROS para el TurtleBot II.

La instalación de PLEXIL necesita programas externos que compilen el código fuente y generen los ejecutables de PLEXIL (g++, ant y java). Se comprueba que dichos programas estén instalados y se procede a la instalación de PLEXIL, descargándo la distribución de su página oficial [50]. Tras su descarga se descomprime en el directorio `/home/usuario/plexil` y se añaden las siguientes variables de entorno:

```
$ export PLEXIL_HOME=/home/usuario/plexil
$ export PATH=$PLEXIL_HOME/bin:$PATH
$ export LD_LIBRARY_PATH=$PLEXIL_HOME/lib
```

Con las variables de entorno añadidas se procede a la compilación de PLEXIL, ubicándose en el directorio `/home/usuario/plexil` y ejecutando los siguientes comandos:

```
$ make clean
$ make all
```

Con esto ya estaría instalado todo el framework de desarrollo PLEXIL.

B.3.2 Modificaciones estructurales

La migración de arquitectura hardware conlleva cambios estructurales en la plataforma física del robot, actuando sobre tres componentes: cámara, pantilt y sónar.

- *Cámara*: se sustituye la Kinect por la Raspberri Pi camera debido a problemas de consumo y rendimiento. Esta nueva cámara ofrece una resolución de 5 megapixels a 1080p y 30 fps, obteniendo capturas de gran resolución. Además se libera un puerto USB ya que su conexión a la placa se hace a través de un conector específico.

- *Pantilt*: se modifica la estructura física de la pantilt para sujetar la nueva cámara, sustituyendo el soporte original por una estrucutra impresa en 3D que contiene en su interior la Raspberry Pi camera junto con unos LED de iluminación y un sensor de ultrasonidos. Asímismo se ha eliminado el Arduino que controlaba la pantilt, puesto que ahora este control se realizará a través de los puertos GPIO incorporados en la Raspberry Pi.

- *Sonar*: los problemas a la hora de detectar obstáculos con los sensores de sonar ha provocado que no se incluyan en esta versión. Sin embargo se ha añadido un sensor de ultrasonidos en la pantilt para proporcionar un feedbak de la distancia a los obstáculos en un escenario teleoperación asistida, sin efecto en la arquitectura de control.

B.3.3 Migración de código

Tras la instalación de todas las herramientas necesarias llega el momento de migrar el código a la Raspberry Pi haciendo una copia desde el repositorio original, sin embargo dicha copia no funcionará en la nueva plataforma hardware debido las modificaciones físicas explicadas en el apartado anterior. Gracias a la modularidad que aporta la arquitectura MoBAr no es necesario modificar todas las capas, sino que únicamente hay que modificar los módulos afectados en la capa funcional:

- *Bloque pantilt*: ahora el bloque actuará directamente sobre los puertos GPIO de la Raspberry Pi, eliminando la librería *rosserial* que comunicaba el Arduino con los topics de ROS. La lectura del topic `/pantilt` es directa, generando inmediatamente la señal PWM en los puertos GPIO conectados a la pantilt.
- *Bloque de visión*: se ha modificado la parte de capturas de imágenes. Se sustituye la captura a través del topic `camera/rgb/image_raw` por la aplicación *raspistill*, quel captura una imagen en el directorio indicado.
- *Bloque sensorial*: se elimina la parte del bloque que recogía la información de los sensores sonar, sin embargo los sensores de parachoque siguen activos para que el robot pueda esquivar obstáculos.

B.4 Resultado final

Tras las modificaciones realizadas, se ha mejorado la estética del robot gracias a la supresión del mini-Pc y a un rediseño de la *pantilt*, junto con la ubicación de la Raspberry Pi en un lugar poco visible. El resultado de este nuevo diseño se muestra en la figura B.1.

Al probar la arquitectura MoBAr en esta nueva plataforma *hardware*, se aprecian ciertos inconvenientes:

- El arranque de la red de ROS pasa de durar 30 segundos a tener una duración media de arranque de 3 minutos.
- La versión de ROS implementada en Raspbian es algo inestable ya que sufre caídas periódicas que obligan a reiniciar toda red manualmente.

Sin embargo cuando todo está arrancado y permanece estable el comportamiento de la arquitectura es similar al comportamiento del modelo anterior (variaciones entre el 1 y el 2 % en distancias y tiempos de ejecución). Además se mejora la autonomía del robot considerablemente, pasando de media hora de autonomía a más de 2,5 horas, debido a la supresión del mini-pc y la Kinect (alto consumo en estático y en torno a 1,5 kg de peso) junto con el bajo consumo de la Raspberry Pi.

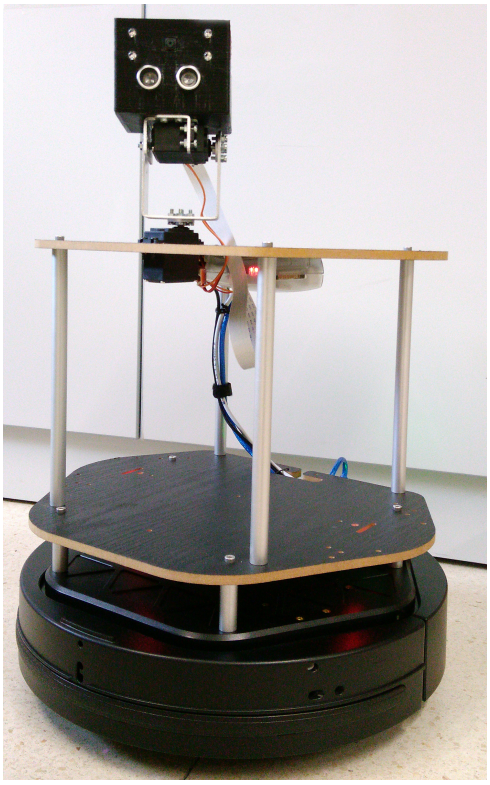


Figura B.1: Turtlebot II controlado con Raspberry Pi

Apéndice C

Herramienta OGATE

Este anexo uestra la herramienta a través de la cual se han realizado todas las pruebas reflejadas en el capítulo 8. El anexo comienza con una descripción de la herramienta en la que se enumeran sus elemntos clave, y termina con el diseño y operabilidad que ofrece esta herramienta para el testeo de arquitecturas de control autónomo.

C.1 Descripción

On-Ground Autonomy Test Enviroment (OGATE) es un proyecto financiado por la ESA cuyo objetivo es ofrecer una herramienta de testeo en el desarrollo de sistemas de contro autónomo, implementando un entorno que provee tareas de verificación y análisis. La infraestrucutura de OGATE se divide en tres boques fundamentales (figura C.1):

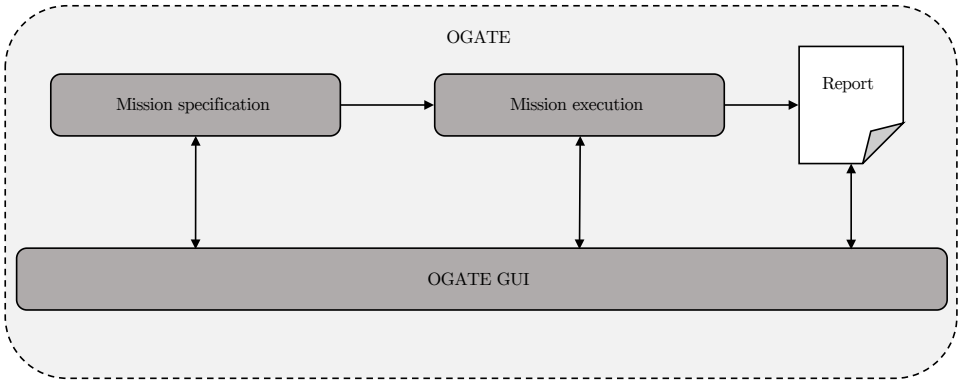


Figura C.1: Infraestructura de OGATE

- **Mission specification:** se encarga de la configuración de la arquitectura de control, así como de establecer las metas a alcanzar. En este bloque se especifican configuraciones básicas necesarias para que la arquitectura de control funcione (parámetros de configuración de las distintas capas) así como las metas que se esperan cumplir.
- **Mission execution:** ejecuta las misiones definidas por el operador, recogiendo métricas de la ejecución. Este bloque lanza la arquitectura de control (MoBAr en nuestro caso) y recoge datos que le permiten al operador tener un feedback de la ejecución del plan.

- **OGATE GUI:** proporciona el soporte gráfico a los módulos anteriores, facilitando la interacción al operador. Gracias a este bloque se puede realizar la configuración de los distintos componentes de forma gráfica, así como obtener en tiempo de ejecución un diagrama de tiempo con las acciones planificadas y la ejecución del plan en tiempo real. Además cuando finaliza el plan obtiene métricas (distancias, energías, tiempo de planificación) y las representa facilitando así su comprensión al operador. También permite integrar plug-ins desarrollados por terceros, permitiendo introducir componentes gráficos específicos dentro del panel de OGATE.

C.2 Diseño y operabilidad

La interfaz gráfica de OGATE presenta una pantalla principal (figura C.2) en la que se encuentran dos secciones bien diferenciadas: una parte superior en la que se encuentran las pestañas de navegación y los botones para iniciar, pausar y detener la ejecución de las arquitecturas de control, y una parte inferior en la que se encuentran los controladores que pueden ser manejadas desde la herramienta. En la figura expuesta aparecen dos controladores de la arquitectura MoBAr con dos planificadores distintos (UP2TA y OPTIC$_{S\theta}$) con los que se realizaron las pruebas del capítulo 8.

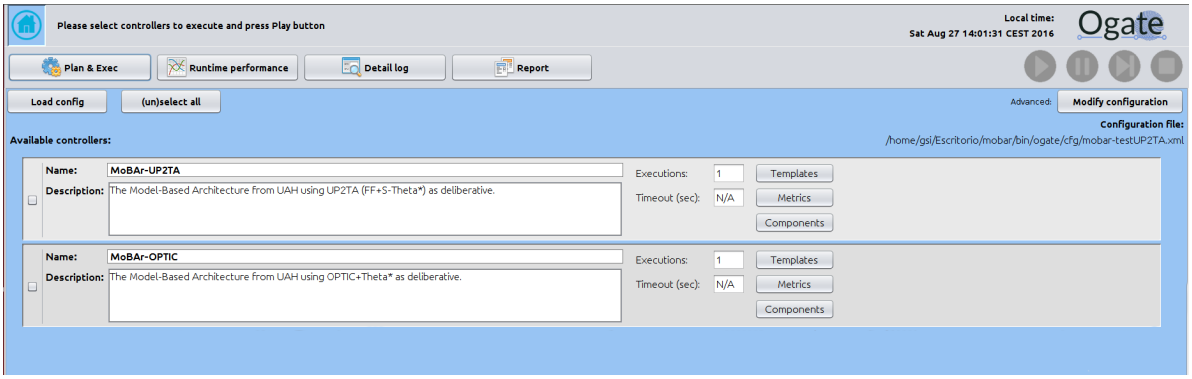


Figura C.2: Pantalla principal de OGATE

Antes de iniciar cualquiera de los dos controladores es necesario verificar o modificar los parámetros de configuración del controlador para realizar las operaciones de forma óptima. Esto también se puede hacer de forma gráfica pulsando sobre el botón *Modify configuration*, obteniéndose la ventana que queda reflejada en la figura C.3. En esta nueva ventana aparecen todas las opciones de configuración para cada uno de los dos controladores. Es posible realizar modificaciones en el dominio y en el problema PDDL, así como configurar el planificador y poner la dirección IP de los servidores RPC implementados en la capa funcional. Además también se pueden configurar las métricas obtenidas tras la ejecución del plan y parámetros de la interfaz gráfica y del ejecutor. Para guardar la configuración permanentemente se pulsa sobre el botón *Save Config*, mientras que si la configuración solo se quiere preservar hasta el cierre OGATE hay que pulsar el botón *Go back to list* el cual retorna a la pantalla principal.

Para iniciar la ejecución controlador, desde la pantalla principal se selecciona el controlador deseado y se pulsa el botón *play*, pasando automáticamente a la pestaña *Plan & Exec*. Esta nueva pestaña tiene componentes que facilitan la visualización y depuración del plan en tiempo de ejecución gracias a las 3 secciones en las que se divide (figura C.4) : (i) en la parte superior se encuentra un timeline en el que se representa la ejecución del plan en tiempo real, o las acciones planificadas (según la pestaña escogida); (ii) en la parte inferior izquierda aparece una representación de los nodos de PLEXIL y su ciclo de ejecución; (iii) en la parte inferior derecha hay una representación del mapa por el que se mueve el robot. Además desde la pestaña (ii) es posible poner puntos de parada en los nodos del plan de PLEXIL y ejecutar el

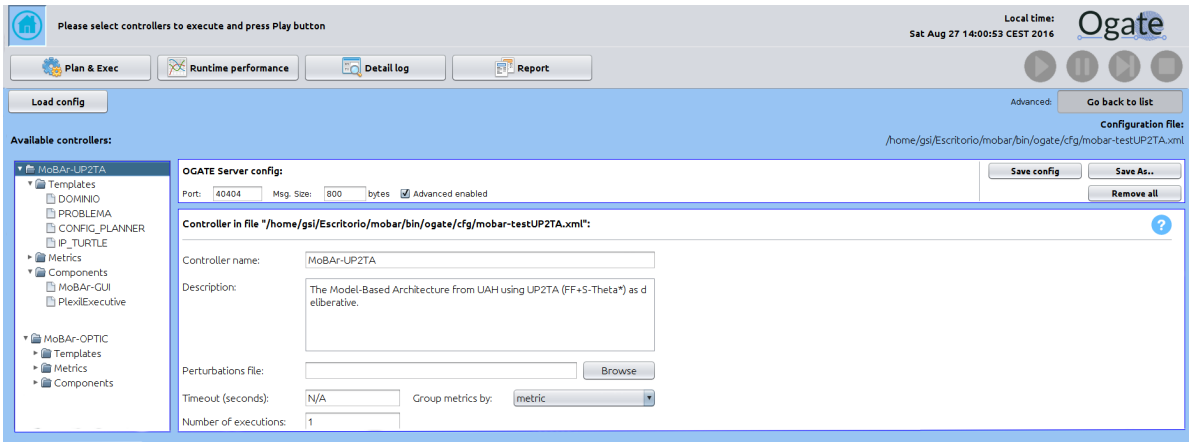


Figura C.3: Ventana de configuración de OGATE

plan paso a paso, depurando cada uno de los movimientos. También es posible visualizar el dominio, el problema PDDL y el plan generado, así como añadir metas al problema PDDL en medio de la ejecución del plan haciendo uso del botón *Add Goal* o marcando una posición en el mapa con el ratón.

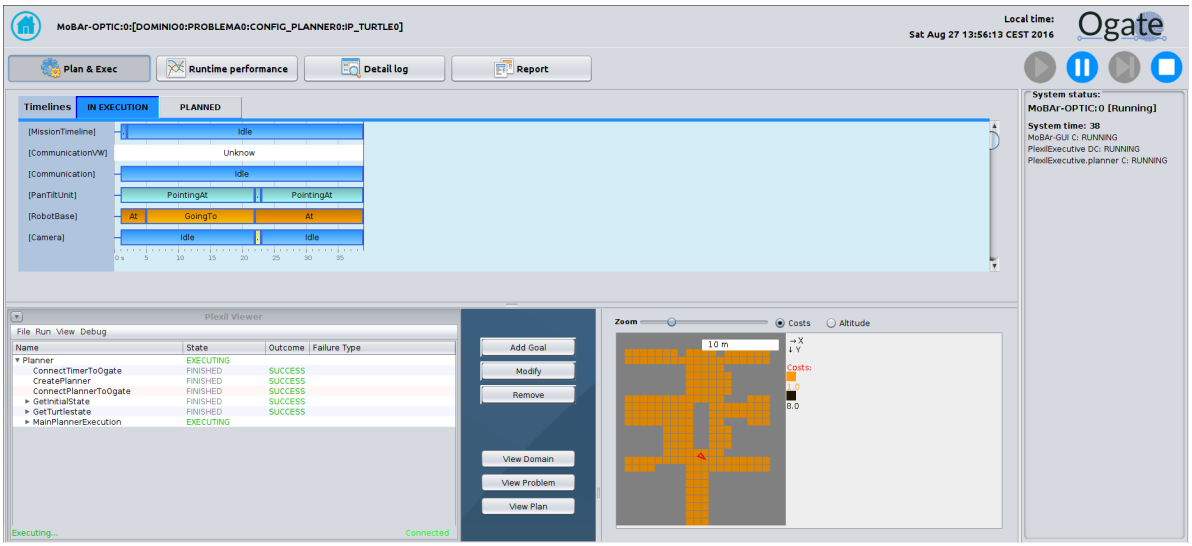


Figura C.4: Ventana *Plan & Exec*

En cuanto al timeline, en figura C.5 se muestra el conjunto de acciones del plan ordenadas secuencialmente según su aparición en el plan, en la que se puede ver como el robot realiza movimientos para recargar su batería y capturar fotografías hasta que finaliza el plan. Sin embargo en la figura C.4 se muestra el mismo plan según se ejecuta, mostrando el estado del planificador, de la pantilt, el movmiento del robot y el estado de la cámara. Cuando el robot realiza algún cambio en alguno de esos componentes aparece reflejado en esa línea de tiempo, teniendo constancia de la duración real de la acción y el resultado de dicha acción.

También es posible obtener distintas medidas de rendimiento en tiempo real desde la pestaña *Runtime perfomance*, obteniendo medidas del tiempo de deliberación, porcentaje de memoria utilizada, porcentaje de procesador consumido, nivel de batería del robot, etc.

La herramienta OGATE también proporciona un seguimiento del estado de PLEXIL (tanto del ciclo de ejecución como del estado de sus adaptadores) en la pestaña *Detail log*. Una muestra de dicha pestaña se puede encontrar en la figura C.6.

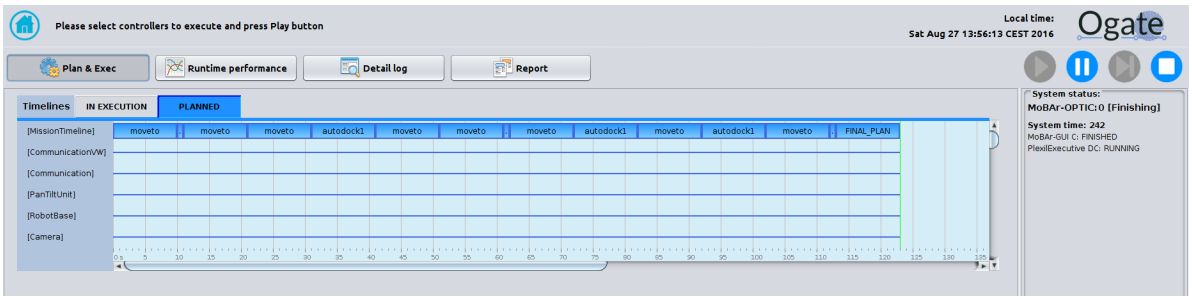


Figura C.5: Timeline del plan generado

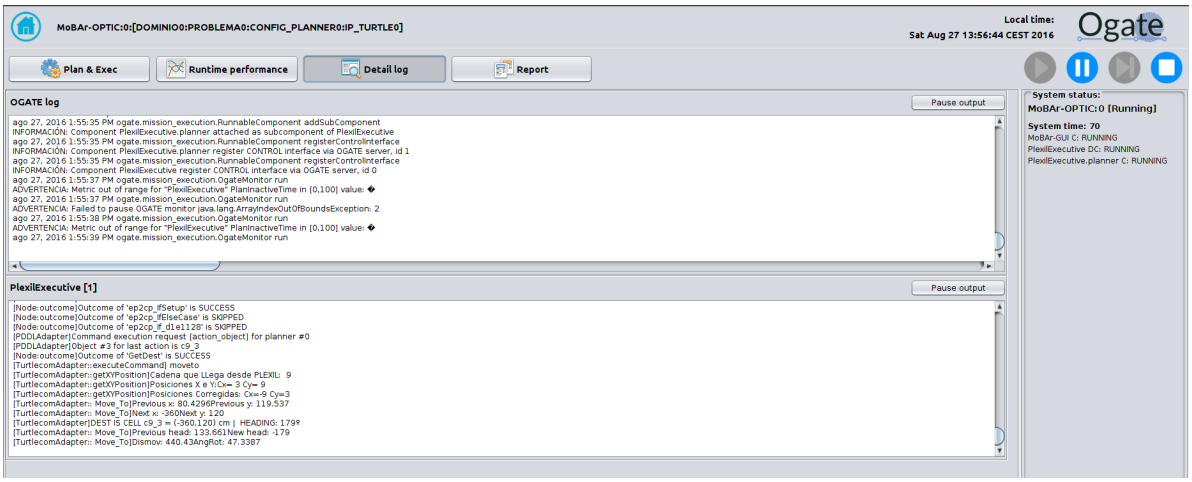


Figura C.6: Log proporcionado por OGATE

Con este log proporcionado por OGATE se pueden hacer labores de depuración a bajo nivel, ya que en él se puede encontrar un registro temporal de cada uno de los pasos que realiza el ejecutor.

Por último la pestaña *Report* proporciona unos gráficos con métricas y valores medios de los datos recogidos para múltiples ejecuciones.

La herramenta OGATE simplifica mucho la vida al operador, ya que visualmente puede recoger grandes cantidades de información.

Apéndice D

Documentación adicional

Este apéndice muestra la documentación acreditadora de que el alumno ha recibido acciones formativas durante la realización del Trabajo Fin de Máster. En primer lugar se muestra la asistencia a un seminario de ROS impartido en la Escuela Politécnica Superior de la Universidad de Alcalá. El segundo apartado contiene la asistencia a un seminario de Matlab sobre Machine Learning y Hardware de bajo coste con Matlab. El tercer apartado contiene la participación en el 8º concurso de ideas para creación de empresasa de base tecnológica de la OTRI. Por último se muestra documentación sobre el curso de verano robótica móvil impartido por el grupo de investigación GSI, en el que el alumno Diego López Pajares participó como profesor.

D.1 Seminario de ROS

En este seminario se impartieron conceptos básicos de entornos de desarrollo robótico. El seminario comenzó mostrando el método de programación clásico en robótica (sin entoros de desarrollo) en el que los programas de ususario interaccionaban directamente con el hardware del robot, para posterioriormente enseñar métodos de programación utilizando entornos de desarrollo, en los que aparece una nueva capa entre el programa de ususario y el entorno de desarrollo que abstrae al usuario del *hardware* de bajo nivel. Esta nueva capa introducida facilita el acceso a los drivers mediante abstracciones de *hardware*, incluyendo además un modelo de programación que facilita la estructuración de *software* y herramientas que facilitan la experiencia de usuario del programador (interfaces gráficas, simuladores, etc.). Una vez introducidos los entornos de desarrollo se introdujo ROS como entorno de desarrollo robótico de *software* libre mostrando sus conceptos claves y alguno de sus simuladores junto al esquema típico de programación en ROS. Por último el seminario incluía una práctica en la que se interacionaba con las herramientas de bajo nivel de ROS y se creaba un robot en el simulador STDR. Para el proyecto el seminario sirvió de apoyo para aumentar los conocimientos en las herramientas de bajo nivel y la creación de archivos `.launch`.

D.2 Seminario de Matlab

La primera parte del seminario de Matlab mostraba las capacidades de Simulink para trabajar con *Machine Learning*, sin demasiado interés para las tareas realizadas en este proyecto. La segunda enseñaba la capacidad de Matlab para interaccionar con msistemas *hardware* de bajo nivel. En primer lugar se mostró el bloque de Simulink Robotic System Toolbox que permite enviar y recibir mensajes a red de

ROS gráficamente. También permite de forma gráfica fijar u obtener valores del servidor de parámetros o enviar mensajes en blanco. Se pueden aprovechar estas herramientas para ajustar de una forma fácil las constantes del algortimo PI implementado en el proyecto. En el seminario también se mostró un paquete que interactúa con el *hardware* de la Raspberry Pi permitiendo ejecutar comandos de la shell, manipular los pines GPIO o trabajar con archivos. Además se mostró en vivo como es posible obtener de forma rápida imágenes de las Raspberry Pi a través de Simulink, para aplicarle técnicas de procesado de señal con el paquete específico de Simulink. Además Matlab también incluye una herramienta para interaccionar con Arduino, facilitando la adquisición analógica y digital de sensores a través de esta plataforma.

D.3 Concurso OTRI

El grupo de investigación GSI se presentó al 8º Concurso de Ideas de la OTRI con su propuesta: *Lares Robotics Solutions: Sistema de teleasistencia avanzado no intrusivo*, que resuelve el problema presente en los servicios de teleasistencia actuales introduciendo un nuevo producto capaz de detectar situaciones de peligro sin la participación activa por parte de la persona dependiente. El producto se compone de 3 bloques de los cuáles uno se corresponde con este proyecto. El concurso consistía en mostrar la viabilidad de una nueva empresa haciendo un plan de empresa a 3 años vista. El pan de empresa incluía un resumen ejecutivo, descripción del producto, equipo de trabajo, plan de marketing, plan de recursos humanos, análisis de riesgos y plan económico-financiero. Con la propuesta el grupo de investigación GSI consiguió el segundo premio del concurso compuesto por un accesit de 1000 € y una estancia en la incubadora de empresas de la UAH.

D.4 Curso de verano

Con el objetivo de dar a conocer los conocimientos adquiridos en el campo de la robótica, en el verano de 2015 el grupo de investigación GSI ofertó un curso de verano de robótica móvil de cinco días de duración en el que se impartieron los siguientes conceptos :

- Día 1: conceptos básicos de robótica. Introducción al mundo de la robótica, tipos de sensores y actuadores, y simuladores robóticos. Incluye una primera toma de contacto con el simulador Gazebo.
- Día 2: sistema operativo robótico. Presentación del framework de desarrollo robótico ROS, mostrando conceptos básicos y herramientas elementales. Al terminar la parte teórica se incluyen ejercicios de laboratorio que ponen en práctica los conocimientos adquiridos en la parte de teoría.
- Día 3: localización, mapeo y SLAM. Introdución a las técnicas de localización robótica y mapeo dentro de un entorno conocido. Tras la explicación se realizan prácticas de odometría haciendo uso de ROS. Por último se introducen las técnicas de localización y mapeo simultáneo (SLAM) para terminar con una práctica/demostración de SLAM en ROS.
- Día 4: Planificación de tareas y rutas. Introducción al mundo de la planificación de tareas y rutas con lenguajes de planificación (PDDL, DDL, ...) con un fuerte base teórica y pequeños ejemplos.
- Día 5: Plataformas robóticas. Visión general de distintas plataformas robóticas en la actualidad, entre las que se encuentran Arduino y Raspberry Pi. Al terminar la parte teórica se hace un práctica que integra ROS en Arduino a través de la librería *rosserial*.

D.4.1 Juego de transparencias

En esta sección se muestra el juego de transparencias elaborado por el alumno Diego López Pajares para la impartición del curso de verano. Dicho alumno se encargó el primer día de introducir las técnicas de control y telemetría junto con los simuladores para posteriormente realizar una práctica de telecontrol en el simulador Gazebo. El segundo día impartió toda la base teórica y práctica de ROS, ocupándose de la parte práctica en el tercer día de curso (prácticas de odometría y SLAM con ROS). Por último el último día colaboró con un compañero del grupo de investigación en la preparación y ejecución de la práctica de integración de ROS en Arduino.

A continuación se muestra el juego de transparencias elaborado por el alumno en orden cronológico de aparición.

Universidad de Alcalá

Teleccontrol y telemetría

Curso de Verano

Robótica móvil

Diego López Pajares

Resumen

- Introducción
- Objetivo
- Rviz
- Gazebo+Rviz

Introducción

- Telecontrol
  - El telecontrol consiste en el envío de órdenes a un sistema remoto a través de un enlace de transmisión (normalmente a través de internet, conexiones radio, …)
  - En robótica surge la necesidad de manejar nuestro robot sin ningún tipo de cables.

Introducción

- Telemetría
  - La telemetría consiste en la medición de magnitudes físicas y su posterior envío al operador del sistema.
  - En robótica es básico conocer el estado de todos los sensores que tiene el robot.
  - El robot que vamos a manejar tiene estos sensores.
    - Encoders
    - Giróscopo
    - Sensor de choque
    - Sensor de caída
    - Cámara

Introducción.

- A modo de recordatorio, veamos los 4 niveles de autonomía de la ESA.

Nivel	Descripción	Funciones
E1	Ejecución de la misión bajo control de tierra	El operador manda un comando al robot y éste lo ejecuta.
E2	Ejecución de operaciones pre-planificadas	El operador manda un comando y el robot lo ejecuta en un tiempo especificado.
E3	Ejecución de misiones adaptativas a bordo	El robot opera en base a unos eventos.
E4	Ejecución de operaciones específicas a bordo.	Misiones orientadas a funciones de replanificación.

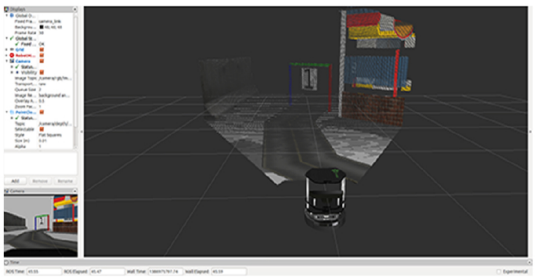
Objetivo

- Ver el funcionamiento de un telecontrol básico mediante el cual se moverá el robot.
- Para hacer la sesión más amena y divertida se va a hacer uso de Gazebo junto a Rviz.

Rviz

- Rviz es una herramienta que permite la representación de los datos que proporciona el robot.
- Para familiarizarnos con el entorno Rviz vamos a lanzar la aplicación
  - roslaunch turtlebot_rviz_launchers view_model.launch

Rviz

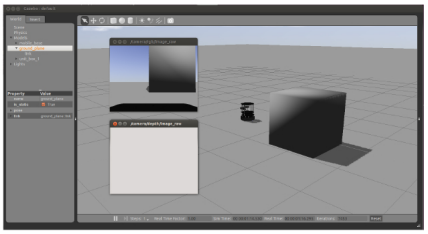


Gazebo + Rviz

- Para que Rviz pueda leer datos de los sensores del robot es necesario tener el robot y leer datos de sus sensores.
- Como no tenemos un robot físico lanzaremos un modelo del robot en Gazebo que simulará el valor de dichos sensores.

Gazebo + Rviz

- El primer paso para lanzar el modelo del robot en Gazebo es ejecutar el siguiente comando en una terminal
  - roslaunch turtlebot_gazebo turtlebot_world.launch



Gazebo + Rviz

- El siguiente paso es ejecutar Rviz para empezar a leer los valores de los sensores que proporciona el robot
- Abriremos una nueva terminal y ejecutaremos el siguiente comando
  - roslaunch turtlebot_rviz_launchers view_robot.launch

Gazebo + Rviz

- Y ya por fin llegó la hora de teleoperar al robot.
- ROS proporciona una aplicación que permite mover el robot con el teclado de nuestro ordenador
- Abriremos una nueva terminal y ejecutaremos el siguiente comando
  - roslaunch turtlebot_teleop keyboard.launch

Gazebo + Rviz.

- Ahora os toca a vosotros interactuar con el entorno añadiendo objetos en Gazebo para ver como afecta en Rviz.

Universidad de Alcalá

Simulación con gazebo.

Curso de Verano

Robótica móvil

Diego López Pajares

Introducción

- Gazebo es un simulador multirobot para entornos exteriores.
- Es capaz de simular numerosos robots, sensores y objetos en un mundo tridimensional.
- Pero, dejémonos de rollo, vamos a jugar un rato!

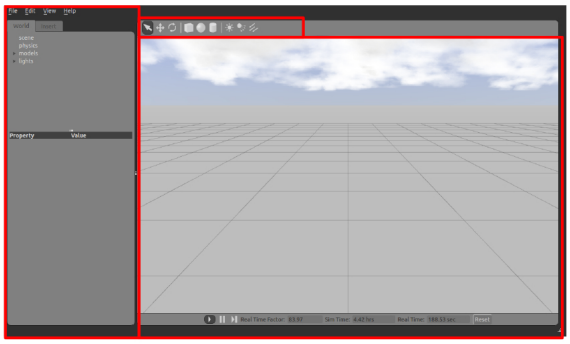
Objetivo

- El objetivo de esta práctica es familiarizarse con el entorno de simulación Gazebo.

Tutorial básico.

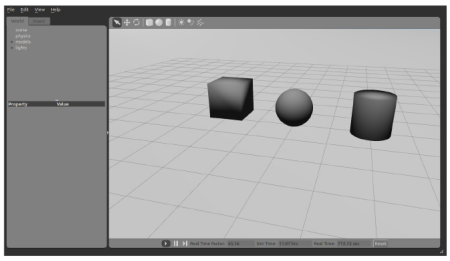
- Lo primero que vamos a hacer es lanzar el simulador, para ello abriremos un nuevo terminal e introduciremos el siguiente comando:
  - gazebo

Tutorial básico.



Tutorial básico.

- Para añadir objetos al mundo basta con seleccionar alguna de las formas de la barra superior y pinchar en el mapa.

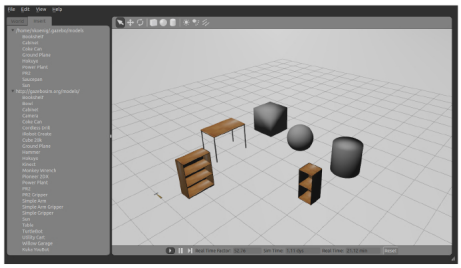


Tutorial básico

- Se puede modificar la posición de los objetos así como su tamaño y su rotación con estas tres herramientas que nos proporciona gazebo:

Tutorial básico

- Además podemos importar objetos más complejos de una base de datos que tiene Gazebo.



Tutorial básico

- Para guardar todos los cambios efectuados en el modelo hay que ir al menú *file* y seleccionar la opción *save world as*.
- El modelo se guarda como un archivo .sdf
- Este archivo .sdf es en realidad un archivo xml en el que se indican todas las características del mundo.

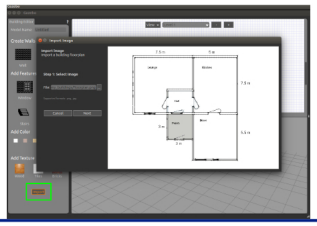
Tutorial básico.

- Además de modelos, Gazebo nos permite crear nuestros propios edificios.
- Para ello nos proporciona la herramienta Building Editor.
- El acceso a la herramienta se encuentra en el menú *edit* (la opción *building editor*)

Tutorial básico

Tutorial básico

- Además en versiones más recientes de Gazebo que la que se va a utilizar en el laboratorio se puede crear un edificio a partir de un plano.
- Para ello basta con importar una imagen del plano en formato .png



UAH 12

Práctica 1

- Crear un mundo que contenga:
  - Un edificio de al menos 4 habitaciones.
  - Que cada habitación tenga al menos un objeto.
  - Los objetos han de ser distintos
  - Debe quedar espacio suficiente para que un robot pueda pasar por todos los sitios (ya que este modelo se intentará utilizar en sesiones posteriores).

UAH 13

Práctica 2

- Modificar el modelo anterior
  - Cambiar los colores,texturas,… de los objetos que se han incluido en el modelo.
  - Cambiar el fondo de la escena para que aparezca un cielo con nubes.
- Para ello habrá que modificar el archivo .sdf en el que está guardado nuestro modelo.

UAH 14

Universidad de Alcalá

Introducción a ROS

Curso de Verano

Robótica móvil

Diego López Pajares.

Resumen

- Introducción
- Instalación de ROS
- Arquitectura ROS
  - Nivel de sistema de ficheros
  - Nivel de computación gráfica
  - Nivel de comunidad
- Comandos ROS
- Práctica

Introducción

- El concepto de Sistema Operativo Robótico (ROS) es un marco que se utiliza comúnmente en robótica.
- Su filosofía es desarrollar un software que pueda ser utilizado en robots de diversa índole haciendo pequeños cambios en el código.
- Para facilitar el desarrollo proporciona una serie de herramientas y librerías.

Introducción

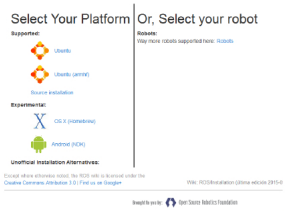
- ROS se desarrolló en 2007 en el laboratorio de inteligencia artificial de Stanford.
- A partir de 2008 este desarrollo continua en el instituto de investigación robótica.
- En la actualidad instituciones de investigación desarrollan proyectos con ROS compartiendo el software.

Introducción

- ROS está en continuo desarrollo, es por eso que aparecen nuevas versiones del sistema operativo cada poco tiempo.
- La versión más reciente de ROS es ROS Jade Turtle, lanzada en marzo de 2015.
- Aunque en este curso se va a trabajar con la versión anterior (ROS Indigo Igloo), cuyo soporte finaliza en abril de 2019.

Instalación de ROS

- El proceso de instalación es muy sencillo.
- Para instalarlo solo hay que ir a la página http://wiki.ros.org/ROS/Installation y escoger el sistema operativo en el que se va a instalar o bien el robot que se va a usar.



Instalación de ROS

- Una vez que se ha seleccionado una de las dos opciones aparece un tutorial que indica todos los pasos necesarios para llevar a cabo la instalación de ROS.
- Debido a la limitación temporal no se va a realizar la instalación de manera presencial, sino que se va a proporcionar una máquina virtual con ROS Indigo Igloo instalado.

UAH 7

Arquitectura ROS

- La arquitectura de ROS está dividida en tres niveles conceptuales:
  - Nivel de sistema de ficheros.
  - Nivel de computación gráfica.
  - Nivel de comunidad.

UAH 8

Arquitectura ROS (sistema ficheros)

- Al igual que un sistema operativo convencional, un programa en ROS está dividido en carpetas y esas carpetas a su vez contienen ficheros con una funcionalidad.

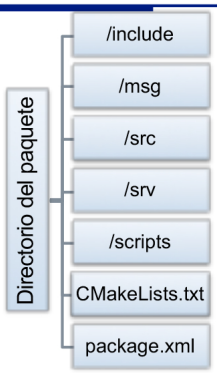
UAH 9

Arquitectura ROS (sistema ficheros)

- Packages
  - Es la principal forma de organización del software en ROS.
  - Pueden contener nodos de ROS, librerías de ROS, archivos de configuración.
  - La estructura de carpetas típica es la siguiente:

UAH 10

Arquitectura ROS (sistema ficheros)



UAH 11

Arquitectura ROS (sistema ficheros)

- Metapackages
  - Es un grupo de paquetes con una funcionalidad específica que son agrupados de forma lógica como un solo paquete.
- Package Manifest
  - Proporciona metadatos sobre un paquete.
    - Nombre, versión, descripción, información de la licencia, dependencias y otra información.
  - Está escrito en formato .xml

UAH 12

Arquitectura ROS (sistema ficheros)

- Message types
  - Definen la estructuras de datos para enviar mensajes en ROS.
  - Esta estructura se almacena en un archivo .msg dentro de la carpeta /msg
  - Esta definición de estructuras de datos facilitan que ROS automáticamente genere código fuente en distintos lenguajes de programación para el tipo de mensaje especificado.

Arquitectura ROS (sistema ficheros)

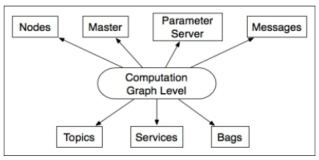
- Service types
  - Define la estructura de los servicios de petición y respuesta en ROS.
  - Esta estructura se define en un archivo con extensión .srv guardado en la carpeta /srv.
  - Al igual que para el caso de los mensajes, estas estructuras facilitan la generación automática de código en ROS.

Arquitectura ROS (sistema ficheros)

- Además ROS proporciona herramientas para movernos por su sistema de ficheros.
- Estas herramientas son similares a los comandos de línea de órdenes cd y ls de linux
  - rosls
  - roscd

Arquitectura ROS (nivel gráfico)

- El nivel de computación gráfica crea una red de procesos interconectados entre sí.
- Los conceptos básicos que se van a ver en este nivel son los siguientes:
  - Nodes
  - Máster
  - Parameter Server
  - Messages
  - Topics
  - Services
  - Bags



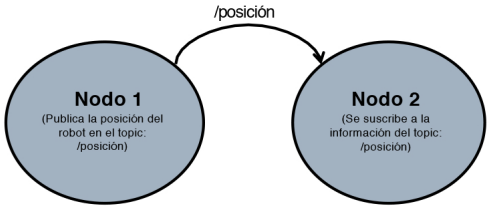
Arquitectura ROS (nivel gráfico)

- Nodos
  - Son elementos de ROS en los que se procesa la información. Los distintos nodos pueden interconectarse entre sí con elementos que se van a ver a continuación (topics, servicies y parameters server).
  - De esta forma se consigue que cada nodo pueda desempeñar funciones específicas como por ejemplo, un nodo controla los motores del robot, otro controla un sensor de choque, …
  - Además se proporciona seguridad adicional al sistema

Arquitectura ROS (nivel gráfico)

- Topics
  - Son buses usados por los nodos para transmitir datos. Cada uno de estos buses (topics) tiene un nombre que lo identifica inequívocamente.
  - Cada topic tiene una estructura fija y muy definida en la que se especifica el tipo y formato de los datos que se transportan.
  - Esta estructura se almacena en un fichero .msg en la carpeta /msg como se vio en el nivel anterior.
  - Los nodos que están interesados en recibir cierta información se suscriben al topic deseado.
  - Los nodos que quieren generar datos y enviarlos publican en el topic deseado.

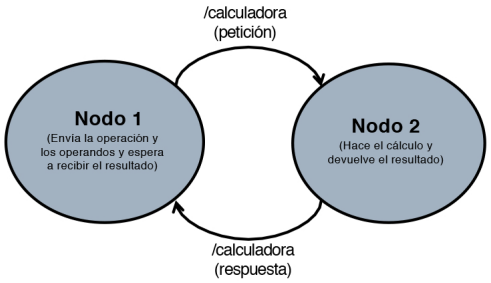
Arquitectura ROS (nivel gráfico)



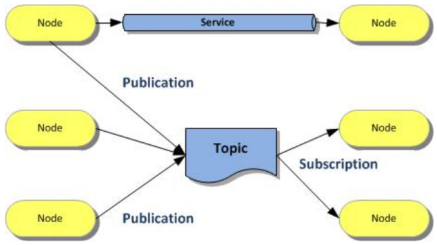
Arquitectura ROS (nivel gráfico)

- Services
  - Cuando un nodo quiere comunicarse con otro nodo y además desea recibir una respuesta se utiliza un servicio.
  - Al igual que en los topics, los mensajes que se intercambian en un servicio están fuertemente tipados.
  - La estructura del mensaje se almacena en un archivo .srv dentro de la carpeta /srv, como se vio en el nivel de sistema de ficheros.

Arquitectura ROS (nivel gráfico)



Arquitectura ROS (nivel gráfico)



Arquitectura ROS (nivel gráfico)

- Messages
  - La comunicación entre nodos realiza mediante un intercambio de mensajes.
  - Un mensaje es una estructura de datos en la que se especifica la organización y el tipo de los datos.
  - Esta estructura de datos se almacena en un fichero .msg dentro de la carpeta /msg.

Arquitectura ROS (nivel gráfico)

- Bags
  - Una bolsa es un fichero creado por ROS con extensión .bag.
  - Se utiliza para guardar toda la información de topics, services, messages y otros.
    - Se suele utilizar para almacenar datos de sensores
  - Esta información se puede visualizar a posteriori para comprobar lo que ha ocurrido.
    - Se puede parar.
    - Rebobinar.
    - Realizar otras operaciones.

Arquitectura ROS (nivel gráfico)

- Master
  - El maestro proporciona servicios de nombre y de registro para el resto de los nodos que coexisten en el sistema ROS.
  - Realiza un seguimiento de los suscriptores y publicadores que hay en los topics y en los services.
  - El papel fundamental del maestro es permitir a los nodos individuales que localicen a otros nodos para que puedan comunicarse.
  - Además el master proporciona el Parameter server
    - Este servidor lo que hace es almacenar datos en tiempo de ejecución. Se suele utilizar para almacenar parámetros de ejecución.

Arquitectura ROS (nivel de comunidad)

- El nivel de comunidad proporciona recursos de distintas comunidades para intercambiar software y conocimientos.
- Los recursos incluyen
  - Distribuciones.
  - Repositorios.
  - La Wiki de ROS
  - Foro.

Comandos ROS

- Para entender mejor estos conceptos, ROS proporciona herramientas que permiten interaccionar con los nodos, servicios, topics,…
- Ejecutar cada uno de los siguientes comandos en un terminal.
  - Roscore
  - Rosrun turtlesim turtlesim_node
  - Rosrun rqt_graph rqt_graph
  - rosrun turtlesim turtle_teleop_key

Comandos ROS

- La herramienta que nos da la información sobre los nodos es rosnode
  - Info
  - Kill
  - List
  - Ping
  - Cleanup

Comandos ROS

- La herramienta que nos da la información sobre los topics es rostopic
  - Bw
  - Echo
  - Find
  - Hz
  - Info
  - List
  - Pub
  - Type

Comandos ROS

- La herramienta que nos da la información sobre los services es rosservice
  - Call
  - Find
  - Info
  - List
  - Type
  - uri

Comandos ROS

- Rosmsg
  - Show
  - List
  - Package
  - Packages
  - Users
  - Md5

Comandos ROS

- Además ROS proporciona una herramienta de visualización que permite tener una perspectiva visual de todos los nodos que se están ejecutando bajo el sistema operativo ROS.
- La herramienta que permite visualizar todo este contenido es rqt_graph
  - Para ejecutarla hay que introducir en un terminal
    - rosrun rqt_graph rqt_graph

Práctica

- Matar el nodo de turtlesim mediante comandos de ROS.
- Identificar el topic en el que se está publicando la velocidad de la tortuga.
- Ver lo que pasa en ese topic cuando se estamos moviendo la tortuga.
- Ver la estructura de mensaje que tiene ese topic
- Publicar en el topic mediante linea de comandos ROS

Universidad de Alcalá

Programación en ROS

Curso de Verano

Robótica móvil

Diego López Pajares

Resumen

- Creación del espacio de trabajo
- Creación de un paquete
- Desarrollo de código

Creación del espacio de trabajo

- Lo primero que hay que hacer antes de desarrollar código en ROS es crear el espacio de trabajo.
- Caktin es el constructor oficial de ROS y es el sucesor del antiguo rosbuild.
- Para crear nuestro espacio de trabajo abrimos una terminal y ejecutamos las siguientes líneas de comandos:
  - Mkdir –p ~/curso_verano/src
  - Cd curso_verano/src
  - Catkin_init_workspace

Creación del espacio de trabajo

- Cd ..
- Catkin_make

- A continuación es necesario incluir nuestro directorio en una variable de entorno de linux para que ROS pueda encontrar todo lo que hay en ella
  - Cd ~
  - gedit ./.bashrc
  - source /home/viki/curso_verano/devel/setup.bash

Creación del paquete

- Una vez que tenemos nuestro directorio de trabajo podemos crear nuestros propios paquetes.
- Vamos a crear un paquete que ejecute un nodo
- Lo primero que hay que hacer al crear el paquete es indicar los paquetes de los que va a depender nuestro nuevo paquete
  - Cd ~/curso_verano_src
  - Catkin_create_pkg nodo_vacio rospy roscpp std_msgs
  - Cd ..
  - Catkin_make

Desarrollo de código

- Una vez que ya se ha creado el paquete ya se puede crear código.
- Como se ha explicado antes los archivos de código fuente irán ubicados en la carpeta /src de nuestro paquete.
- Para poder compilar el código es necesario modificar el archivo CmakeList.txt

Universidad de Alcalá

Práctica odometría

Curso de Verano

Robótica móvil

Diego López Pajares

Resumen

- Objetivos
- Desarrollo de la práctica

Objetivos

- En esta práctica se pretende entender el funcionamiento de la odometría del robot.
- En un primer lugar se analizará la odometría del robot a través de los comandos que vimos ayer en ROS.
- Posteriormente crearemos nuestro propio programa que le indicará al robot la distancia que tiene que avanzar o el ángulo de rotación que tiene que girar.

Desarrollo de la práctica

- Lo primero que vamos a hacer es lanzar nuestro robot en el entorno de simulación gazebo
  - roslaunch turtlebot_gazebo turtlebot_world.launch
- Una vez que hemos lanzado el modelo vamos a ver en que topic publica la odometría
  - rostopic list
  - rostopic echo /…
- También vamos a analizar la estructura del mensaje
  - rostopic type /…
  - Romsg show /….

Desarrollo de la práctica

- Una vez que hemos entendido dónde y cómo publica la odometría el robot vamos a crear nuestro propio paquete.
  - cd curso_verano/src
  - catkin_create_pkg odometria rospy roscpp geometry_msgs std_msgs
  - cd ..
  - catkin_make

Desarrollo de la práctica

- Una vez que tenemos ya creado nuestro paquete, lo siguiente que hay que realizar es
  - Incluir nuestro código fuente en la carpeta /src de nuestro paquete
  - Modificar el archivo Cmakelist.txt para compilar el programa

Desarrollo de la práctica

- Si todo funciona correctamente, el código se compilará en un instante y tendremos listo un programa que actúa sobre la odometría del robot.

Universidad de Alcalá

Práctica SLAM

Curso de Verano

Robótica móvil

Diego López Pajares

Resumen

- Objetivos
- Desarrollo de la práctica

Objetivos

- En esta práctica se pretende ver como funciona SLAM integrado en ROS.
- Para ello se va hacer una demostración con el robot físico

Introducción

- Por suerte para nosotros SLAM viene integrado dentro de ROS y no hay que hacer todos los cálculos que se han explicado en la teoría.
- Para que funcione solamente hay que lanzar la aplicación y el sistema se encarga de hacer automáticamente el mapeo del entorno.

Desarrollo de la práctica

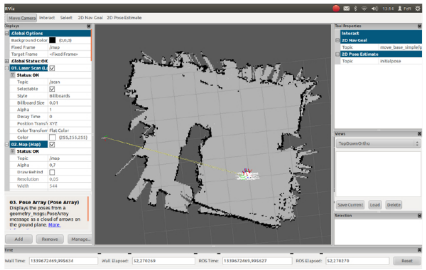
- Lo primero que hay que hacer en el caso de tener el robot físico es lanzar todos los nodos necesarios para que la tortuga opere correctamente.
  - roslaunch turtlebot_bringup minimal.launch
- En el caso de no disponer de un robot físico se hará uso de Gazebo
  - roslaunch turtlebot_gazebo turtlebot_world.launch

Desarrollo de la práctica

- El siguiente paso es ejecutar la aplicación que nos permite mapear con SLAM
  - En el caso de tener un robot físico
    - Roslaunch turtlebot_navigation gmapping_demo.launch
  - En el caso de no disponer de un robot_físico
    - roslaunch turtlebot_gazebo gmapping_demo.launch

Desarrollo de la práctica

- Para poder ver los resultados de la exploración se va a utilizar Rviz
  - roslaunch turtlebot_rviz _launchers view_navigation.launch



Desarrollo de la práctica

- Ya solo queda teleoperar al robot para que vaya construyendo el mundo según lo exploremos.
- La teleoperación se realizará con la teclas del teclado y la aplicación que nos permite realizar está acción es la siguiente
  - roslaunch turtlebot_teleop keyboard.launch
- Con esto ya podremos mapear el entorno que queramos como locos.

Desarrollo de la práctica

- Una vez que se ha construido el mapa solo hay que guardarlo
- Para guardar el mapa hay que ejecutar el siguiente comando
  - rosrun map_server map_saver -f /*directorio_deseado*
- Nota: Es recomendable guardar el mapa antes de cerrar ninguna de las aplicaciones anteriores, si no se perderá el mapa

UAH 9

Desarrollo de la práctica

- Por último sólo quedar probar la navegación del robot dentro de nuestro mapa.
- ROS proporciona una aplicación en la que el robot es capaz de navegar de manera autónoma de un punto a otro del mapa.
- Además si le introducimos nuevos obstáculos será capaz de esquivarlos.

UAH 10

Desarrollo de la práctica

- La aplicación que hay que lanzar es la siguiente
  - En el caso de tener un robot físico
    - Roslaunch turtlebot_navigation amcl_demo.launch map_file:=/tmp/my_map.yaml
  - En el caso de no disponer de un robot_físico
    - roslaunch turtlebot_gazebo amcl_demo.launch map_file:=/tmp/my_map.yaml

UAH 11

Desarrollo de la práctica

- Y con esto se ha acabado la práctica de SLAM.

UAH 12

Bibliografía

[1] P. Aschwanden, V. Baskaran, S. Bernardini, C. Fry, M. D. R-Moreno, N. Muscettola, C. Plaunt, D. Rijsman, and P. Tompkins, "Model-unified planning and execution for distributed autonomous system control," in *Procs. of Association for the Advancement of Artificial Intelligence (AAAI) 2006 Fall Symposia*, Washington DC, USA, October 2006.

[2] A. Ceballos, S. Bensalem, A. Cesta, L. D. Silva, S. Fratini, F. Ingrand, J. Ocón, A. Orlandini, F. Py, K. Rajan, R. Rasconi, and M. V. Winnendael, "A Goal-Oriented Autonomous Controller for Space Exploration," in *Procs. of11th Symposium on Advanced Space Technologies in Robotics and Automation*, Noordwijk, the Netherlands, April 2011.

[3] P. Muñoz and M. D. R-Moreno, "Deliberative Systems for Autonomous Robotics: A brief comparision betwen action-oriented and timelines-based approaches," in *Procs. of ICAPS Workshop on Planning and Robotics*, Rome, Italy, June 2013.

[4] P. Muñoz, M. D. R-Moreno, and A. Martınez, "A first approach for the autonomy of the exomars rover using a 3-tier architecture," in *Procs. of the 11th ESA Symposium on Advanced Space Technologies for Robotics and Automation, Noordwijk, The Netherlands*, 2011.

[5] P. Muñoz and M. D. R-Moreno, "Model-Based Architecture on the ESA DROV simulator," in *Procs. of 23rd ICAPS Application Showcase*, Rome, Italy, June 2013.

[6] M. Taddei, *Leonardo da Vinciś robots*. Hillsdale, NJ: Leonardo3, 2007.

[7] J. Devol, "Programmed article transfer," 1961, uS Patent 2,988,237.

[8] R. R. Murphy, *An Introduction to AI Robotics*, MIT, Ed. MIT, 2000.

[9] R. A. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, vol. 2, pp. 14–23, 1986.

[10] N. J. Nilsson, "A mobile automation: an application of artificial intelligence techniques," in *Procs. of 1st International Joint Conference on Artificial Intelligence*, Washington, D.C, USA, May 1969.

[11] K. Z. Haigh and M. M. Veloso, "Interleaving planning and robot execution for asynchronous user requests," *Autonomous Robots - Special Issue on Autonomous Agents*, vol. 5, no. 1, pp. 79–95, 1998.

[12] J. E. Laird, "Extending the Soar cognitive architecture," in *Procs. of Conference on Artificial General Intelligence*, Memphis, Tennessee, USA, March 2008.

[13] R. Lumia, J. Fiala, and A. Wavering, "The NASREM robot control system standard," *Robotics and Robotics-Integrated Manufacturing, special issue on Robots in Manufacturing*, vol. 6, no. 4, pp. 303–308, 1989.

[14] R. C. Arkin, "Integrating behavioral, perceptual, and world knowledge in reactive navigation," *Robotics and Autonomous Systems*, vol. 6, no. 1-2, pp. 105–122, 1990.

[15] E. Gat, "Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots," in *Procs. of the Tenth National Conference on Artificial Intelligence (AAAI)*, San Jose, CA, USA, July 1992, pp. 809–815.

[16] R. P. Bonasso, R. J. Firby, E. Gat, D. Kortenkamp, D. P. Miller, and M. G. Slack, "Experiences with an architecture for intelligent reactive agents," *Experimental and Theoretical Artificial Intelligence*, vol. 9, no. 2, pp. 187–202, 1997.

[17] K. Konolige and K. Myers, "The Saphira architecture for autonomous mobile robots," MIT Press, Tech. Rep., November 1996.

[18] H. Bruyninckx, "Open robot control software: the OROCOS project," in *Procs. of 2001 ICRA. IEEE International Conference*, vol. 3. IEEE, 2001.

[19] R. Smits, "KDL: Kinematics and Dynamics Library," http://www.orocos.org/kdl.

[20] K. Gadeyne, "BFL: Bayesian Filtering Library," http://www.orocos.org/bfl.

[21] P. Soetens, "RTT: Real-Time Toolkit," http://www.orocos.org/rtt.

[22] "Página oficial de Rock, the Robot Construction Kit," http://rock-robotics.org/stable/index.html.

[23] P. M. Newman, "Moos-mission orientated operating suite," *Massachusetts Institute of Technology, Tech. Rep*, vol. 2299, no. 08, 2008.

[24] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *Procs of ICRA Workshop on Open Source Software*, 2009.

[25] Y. Kusuda, "Robots at the international robot exhibition 2011 in tokyo," *Industrial Robot: An International Journal*, vol. 39, no. 3, pp. 231–235, 2012.

[26] K. Hashimoto, F. Saito, T. Yamamoto, and K. Ikeda, "A field study of the human support robot in the home environment," in *Procs. of 2013 IEEE Workshop on Advanced Robotics and its Social Impacts*. IEEE, 2013.

[27] T. Mukai, S. Hirano, H. Nakashima, Y. Kato, Y. Sakaida, S. Guo, and S. Hosoe, "Development of a nursing-care assistant robot riba that can lift a human in its arms," in *Procs. of 2010 IEEE/RSJ International Conference on*. IEEE, 2010.

[28] S. Coradeschi, A. Cesta, G. Cortellessa, L. Coraci, J. Gonzalez, L. Karlsson, F. Furfari, A. Loutfi, A. Orlandini, F. Palumbo *et al.*, "Giraffplus: Combining social interaction and long term monitoring for promoting independent living," in *Procs. of 6th International Conference on Human System Interactions (HSI)*, 2013.

[29] S. Coradeschi, A. Cesta, G. Cortellessa, L. Coraci, C. Galindo, J. Gonzalez, L. Karlsson, A. Forsberg, S. Frennert, F. Furfari *et al.*, "Giraffplus: a system for monitoring activities and physiological parameters and promoting social interaction for elderly," in *Human-Computer Systems Interaction: Backgrounds and Applications 3*. Springer, 2014, pp. 261–271.

[30] M. Nakamura, M. Sakiyama, J. Suzurikawa, S. Tsukada, A. Ohta, Y. Kume, H. Kawakami, K. Inoue, and T. Inoue, "Methodology for user and user's life centered clinical evaluation of assistive technology (ulceat): Evaluation with prototype roboticbed®," *Technology and Disability*, vol. 24, no. 4, pp. 273–282, 2012.

[31] L. Palopoli, A. Argyros, J. Birchbauer, A. Colombo, D. Fontanelli, A. Legay, A. Garulli, A. Giannitrapani, D. Macii, F. Moro *et al.*, "Navigation assistance and guidance of older adults across complex public spaces: the dali approach," *Intelligent Service Robotics*, vol. 8, no. 2, pp. 77–92, 2015.

[32] "Página web robot Roomba," http://www.irobot.es/robots-domesticos/aspiracion.

[33] "Página de robots que usan ROS," wiki.ros.org/Robots.

[34] J. G. Ziegler and N. B. Nichols, "Optimum settings for automatic controllers," *trans. ASME*, vol. 64, no. 11, 1942.

[35] V. Verma, A. Jónsson, C. Pasareanu, and M.Iatauro, "Universal Executive and PLEXIL: Engine and Languaje for Robust Spacecraft Control and Operations," *American Institute os Aeronautics and Astronautics Space*, 2013.

[36] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL-the planning domain definition language," *International Planning Competition*, 1998.

[37] M. Fox and D. Long, "PDDL2. 1: An Extension to PDDL for Expressing Temporal Planning Domains." *Journal of Artificial Intelligence Research(JAIR)*, vol. 20, pp. 61–124, 2003.

[38] S. Edelkamp and J. Hoffmann, "PDDL2. 2: The language for the classical part of the 4th international planning competition," *4th International Planning Competition (IPC'04), held at ICAPS'04*, 2004.

[39] A. Gerevini and D. Long, "Plan constraints and preferences in PDDL3," *The Language of the Fifth International Planning Competition. Tech. Rep. , Department of Electronics for Automation, University of Brescia, Italy*, vol. 75, 2005.

[40] P. Muñoz, M. D. R-Moreno, and D. F. Barrero, "Unified framework for path-planning and task-planning for autonomous robots," *Robotics and Autonomous Systems*, vol. 82, pp. 1–14, 2016.

[41] P. Muñoz and M. D. R-Moreno, "S-Theta*: low steering path-planning algorithm," in *SGAI*. Springer, 2012, pp. 109–121.

[42] K. Daniel, A. Nash, S. Koenig, and A. Felner, "Theta*: Any-angle path planning on grids," *Journal of Artificial Intelligence Research*, vol. 39, pp. 533–579, 2010.

[43] J. Benton, A. J. Coles, and A. Coles, "Temporal Planning with Preferences and Time-Dependent Continuous Costs." in *Procs. of the International Conference on Automated Planning and Scheduling*, Atibaia, So Paulo, Brazil, 2012.

[44] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, 2001.

[45] "Integración de sensores SHARP en la base Kobuki," http://kobuki.yujinrobot.com/wiki/ir-sensors/ [Último acceso 25-07-2016].

[46] "Página de instalacion de ROS," http://wiki.ros.org/indigo/Installation/Ubuntu.

[47] "Página de instalación paquetes TurtleBot ," http://wiki.ros.org/turtlebot/Tutorials/indigo/Turtlebot%20Installation.

[48] "Página Web oficial de Raspbian," https://www.raspbian.org.

[49] "Página Web de descarga de Rapbian," https://www.raspberrypi.org/downloads/raspbian/.

[50] "Página oficial de descarga de PLEXIL," https://sourceforge.net/projects/plexil/.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR